A Fully Abstract Semantics for a Functional Language with Logic Variables

Radha Jagadeesan Prakash Panangaden Keshav Pingali
Computer Science Department
Cornell University
February 23, 1989

Abstract

There is much interest in the declarative languages community in integrating logic variables into functional languages. In this paper, we give a full semantic account of such a language. We present a Plotkin-style operational semantics for the language and an abstract semantics that expresses meanings as closure operators on a Scott domain. We also show that the denotational semantics is fully abstract with respect to the operational semantics.

1 Introduction

In this paper, we present a novel denotational semantics for a functional language with logic variables and we show that it is fully abstract with respect to a structural operational semantics. This work incorporates three contributions. First, we integrate updatable arrays into the purely functional, dataflow language Id [7] by using logic variables. Second, we develop a denotational semantics that embodies the idea that one can view definitions of a logic variable as imposing constraints on its value. The viewpoint of constraints is well-known to the logic programming community [2]. We define a semantics that expresses the meanings of expressions as closure operators [11] on a certain Scott domain. The denotational semantics provides a much more useful view of programs than does the operational semantics since the language is intended for parallel execution and operational reasoning may involve complex interleavings of execution threads. For our denotational semantics to be meaningful, it must correspond sensibly to an operational semantics. We formalize the intended execution model as a Plotkin-style, structural operational semantics 9 and show full abstraction 8. This is rather pleasing since the denotational semantics is based on rather different ideas than the operational semantics.

Our interest in integrating logic variables into a functional language arises from the well-known fact that incremental definition of arrays in a pure functional language is inefficient since it can lead to a lot of copying. Logic variables provide an elegant solution to this problem; in an array of logic variables, the logic variables can be bound incrementally in the program. These observations motivated the design of Id Nouveau, which is a functional language with arrays that behave like first-order terms in logic programming languages[7]. In this paper, we provide a formal description of a first-order subset of Id Nouveau called Cid.

The semantic account presented here leads to a better understanding of what it means to combine functional and logic programming. Most accounts of languages that combine functional and logic programming have concentrated on the operational semantics, for example QUTE [10]. Lindstrom's language, FGL+LV, has been given both types of semantics but the denotational semantics explicitly incorporates operational notions such as demand signals and data tokens[4]. Our denotational semantics is formulated somewhat more abstractly in terms of equation solving. This is pleasing conceptually and also provides an abstract way of thinking about parallel execution of such programs. We omit proofs and detailed discussions from this paper and refer the interested reader to a companion technical report for details[3].

2 Informal Introduction to the Language

To augment a functional language with logical arrays, we introduce three constructs for allocating, storing into and reading from arrays. An array with uninitialized logic variables as its elements is allocated by the expression array(e)
where e is an integer-valued expression specifying the size of the array. Array updating is performed by a definition like A[i] = v. The value v is unified with the value contained in A[i] and the resulting value is stored into A[i]. Thus, if A[i] was undefined (i.e., it was an uninitialized logic variable), the execution of this definition results in the value v being stored in A[i]. If unification fails, the entire program is considered to be in error. An element of an array may be selected by A[i]. We permit an uninitialized variable to be returned as the result of executing a program. Here is a simple Id Nouveau program:

```plaintext
{k = array(10);
 A[1] = 2;
 fill-even(A,5);
 fill-odd(A,5);
 in A}

def fill-even(X,h) = {for i from 1 to h do
  X[2*i] = X[2*i-1]*2 od
}
def fill-odd(X,h) = {for i from 1 to h do
}
```

When executed on a dataflow simulator, this program produces an array of length 10 in which the i'th element is 2. Procedure fill-even fills in the even elements of array A by reading the odd elements and multiplying them by 2 and procedure fill-odd works similarly. An attempt to execute this program sequentially would lead to incorrect results. One must interleave the execution sequences. Fortunately, the viewpoint of constraints provides a nice way to mask these parts of the procedure. The value of the expression is returned as the result of procedure execution:

\[ \text{expression} = \text{constant | id | expr op expr | cond(expr, expr, expr) | array(expr) | expr[expr] | F(expr)} \]

Figure 1: Syntax of Cid

x \parallel y \parallel e). To sidestep issues regarding the scopes of variables, we follow the logic programming convention: the body of the procedure is a single scope and the formal parameters of the procedure are in the same scope.

3 Operational Semantics of Cid

A configuration is a quadruple \(\langle D, \text{expr}, \rho, \text{FL} \rangle\) where \(D\) is a set of definitions, expr is an expression, \(\rho\) is the syntactic environment and \(\text{FL}\) is the free-list. Intuitively, \(D\) contains definitions whose right-hand sides have not yet been completely 'reduced' to a base value - that is, an identifier, constant or array. The syntactic environment \(\rho\) is a non-empty set of alias-sets where an alias-set is an equivalence class of base values. For example, \(\{x,y,z\}, \{x,y,1\}\) and \(\{x,y,L1,L2\}\) are alias-sets. If \(b1\) and \(b2\) are two base values in the same alias-set, then occurrences of \(b1\) in \(D\) and \(e\) may be replaced by \(b2\) without changing the meaning of the program.

Configurations are rewritten by reduction and by constraint solving. Once the right-hand side of a definition in \(D\) has been reduced to a base value, the definition is removed from \(D\) and unified with the environment. If unification fails, the configuration is rewritten to 'Error' and computation aborts. Otherwise, the resulting environment replaces the old one in the configuration, and rewriting continues. We define some syntactic categories required for the operational semantics.

\[ z, L, \text{id} = \text{countable set of identifiers} \]
\[ \text{cc} \quad \text{constant} = \text{set of constants} \]
\[ \text{Arr} \quad \text{Array} = \{x1, \ldots, xk\} \]
\[ Bv \quad \text{Base-value} = z \in \text{Ar} \]
An Abstract Semantics for Cid

The way to think about Cid programs is in terms of constraints. Thus a definition of the form $x = e$ is viewed as providing a constraint on $x$. Given this view, the meaning of a constraint is the set of values satisfying the constraint. Each time unification is performed new constraints are imposed on some variables. This always adds information, thus we describe a constraint via a function that adds to the "information content" of its argument. Such functions are just sets over functions, i.e. functions that satisfy...
Expressions:

Identifiers: 1. $D, x, p, FL$  \rightarrow  $D, \rho(x), p, FL$ (if $\rho(x)$ is defined)

Basic Operations:

1. $D, e_1, e_2, p, FL$  \rightarrow  $D', e_1 \cdot e_2, p, FL'$
2. $D, e_1 \cdot e_2, p, FL$  \rightarrow  $D', e_1 \cdot e_2, p, FL'$ (where $r = m \circ n$)

Conditional:

1. $D, e_1, p, FL$  \rightarrow  $D', e_1 \cdot p, FL'$
2. $D, e_1 \cdot p, p, FL$  \rightarrow  $D, e_1 \cdot p, p, FL$

Array:

1. $D, e_1, p, FL$  \rightarrow  $D', e_1 \cdot p, FL'$
2. $D, e_1 \cdot p, p, FL$  \rightarrow  $D, e_1 \cdot p, p, FL$

Array Selection:

1. $D, e_1, p, FL$  \rightarrow  $D', e_1 \cdot p, FL'$
2. $D, e_1 \cdot p, p, FL$  \rightarrow  $D, e_1 \cdot p, p, FL$

Application:

1. $D, e_1, p, FL$  \rightarrow  $D', e_1 \cdot p, FL'$
2. $D, e_1 \cdot p, p, FL$  \rightarrow  $D, e_1 \cdot p, p, FL$

Definitions:

1. $D, e_1, p, FL$  \rightarrow  $D', e_1 \cdot p, FL'$
2. $D, e_1 \cdot p, p, FL$  \rightarrow  $D, e_1 \cdot p, p, FL'$ (if $\mathcal{U}(\rho, \{x, y\})$ is consistent)
3. $D, e_1 \cdot p, p, FL$  \rightarrow  $D, e_1 \cdot p, p, FL'$ (if $\mathcal{U}(\rho, \{x, c\})$ is consistent)
4. $D, e_1 \cdot p, p, FL$  \rightarrow  $D, e_1 \cdot p, p, FL'$ (if $\mathcal{U}(\rho, \{x, L_1, \ldots, L_n\})$ is consistent)

Figure 2: Operational Semantics of Cid
\( \forall x. x \in f(x) \). Clearly we want these functions to be monotonic and continuous as well, since the process of generating constraints is supposed to be computable. A final natural requirement is that the functions be idempotent.

**Definition 2** A closure operator, \( f \), on a domain \( V \) is a continuous function satisfying, (i) \( \forall x \in V. x \subseteq f(x) \), (ii) \( f \circ f = f \).

The set of fixed points of a closure operator, \( f \), on a domain \( D \), i.e., the set \( \{ f(x) \mid x \in D \} \), is the set of values that satisfy the constraint \( x = f(x) \).

The least closure operator on a domain is the identity function. The collection of closure operators themselves forms a complete partial order. One can find the least common fixed point of any finite number of closure operators.

**Lemma 1** Suppose \( f \) and \( g \) are closure operators on \( D \). The function \( f \circ g \) is also extensive, continuous and monotonic and the least fixed point of \( f \circ g \) is the least common fixed point of \( f \) and \( g \).

The domain of values that we use is the domain of nested arrays. To define the domain of arrays we use a standard construction \([5]\) for defining a domain of (possibly infinite) terms in logic programming. First, we need some notation. Let \( \omega \) be the set of natural numbers. We use \( \omega^* \) for the set of finite sequences of integers. A sequence is written \([i_1, \ldots, i_n]\). If \( s \) and \( t \) are sequences, then \([s, t]\) denotes their concatenation. If \( s \) is a sequence and \( n \) is a natural number, then \([s, n]\) is the sequence \( s \) with \( n \) added at the end. The size of a set \( X \) is written \( |X| \) and the size of a sequence \( s \) is written \( |s| \).

**Definition 3** A tree \( T \) is a subset of \( \omega^* \) satisfying

1. \( \forall s \in \omega^* \) and \( \forall j \in \omega \), we have \( ([s, i] \in T \land i < j) 
\qquad = (s \in T \land [s, j] \in T) \)
2. \( \{[i][s, i] \in T\} \) is finite for all \( s \in T \)

These define finitely branching trees that may be infinitely deeply nested. The sequences are the tree addresses of the nodes of the tree. Define \( br(s, t) \) to be the number of successors of the node \( s \) in the tree \( t \). If this number is \( 0 \), we have a leaf.

The domain \( V \) is defined in two stages. First, we define a domain \( W \) and then add a top element \( \top \). Let \( \text{Atom} \) be a given domain of atomic values, and let \( \text{Arrays} \) be the set of array constructors written as \{array1, array2, \ldots\}. Let \( A = \text{Atom} \cup \{\top\} \cup \text{Arrays} \), where \( \Omega \) stands for the undefined element.

**Definition 4** An element of \( W \) is a function \( f : t \rightarrow A \) where \( t \) is a non-empty tree. The function \( f \) satisfies \( \forall x \in t. br(s, t) = 0 = f(s) \in (\text{Array}(\Omega) \cup \text{br}(s, t) = n \neq 0 \Rightarrow f(s) = \text{array}_n \). The ordering between the elements of \( W \) is defined as follows: \( f \subseteq g \) if \( \text{dom}(f) \subseteq \text{dom}(g) \) and \( \forall s \in \text{dom}(f). \text{br}(s, \text{dom}(f)) \neq 0 \Rightarrow \text{br}(s, \text{dom}(g)) = \text{br}(s, \text{dom}(f)) = 0 \land f(s) \neq \Omega \Rightarrow g(s) = f(s) \).

Note that if two arrays have different widths, they are incomparable. It is straightforward to check that \( V \) is an algebraic lattice.

To illustrate informally the role of closure operators in the denotational semantics, let us consider a small example. Consider the definitions

\[ x = \text{array}(2) \]
\[ x[1] = 1 \]
\[ x[2] = 2 \]

These may be part of a Cid program. We may view these definitions as imposing constraints on \( x \). The first equation says that \( x \) is an array of size 2. The set of arrays of size 2 (together with \( \top \)) can be represented as the fixpoints of a closure operator on the domain \( V \) as follows. Let us call the array \([1, 2]\), which is the least array of size 2, \( \bot_2 \). Now the desired closure operator is just \( \lambda u. u \uparrow \bot_2 \). Similarly the closure operators representing the next two constraints are \( \lambda u. u \cup [1, 1] \) and \( \lambda u. u \cup [1, 2] \). The composite of these three functions is \( \lambda u. u \cup \bot_1 \). Clearly the least fixed point of the composite function is \([1, 2]\). A less trivial example is obtained by having \( x = \text{array}(2), x[1] = x[2] \). The closure operator representing these two constraints is \( \lambda u. \text{let } v = u[1] \uparrow u[2] \text{ in } [v, v] \). To obtain an element of the set of values that satisfy this constraint, we supply an approximation, say \([a, b]\), and we will get as result \([a \uparrow b, a \uparrow b]\), which clearly satisfies the constraint.

In this spirit, we will let the meaning of an expression be a closure operator of type \( (\text{ENV} \times V) \rightarrow (\text{ENV} \times V) \) where \( \text{ENV} \) is \( \text{Id} \). One may read these meanings in the following way. Each expression is supplied with an environment and an estimate of the resulting value. The semantics refines the environment to incorporate the effect of any new constraints that may result from the evaluation of the expression, and refines the value supplied. We write the curried semantic function \( \text{E} : \text{Exp} \rightarrow (\text{ENV} \rightarrow V) \rightarrow (\text{ENV} \times V) \) where the subscript on the arrow signifies the domain of closure operators. We assume that the environment contains bindings for function names as well. Strictly speaking then, an environment is the sum of an ordinary environment and a functional environment.
\[\mathcal{E}[\text{const}] \; \text{env} \; a = \langle \text{env}, \mathcal{K}(\text{const}) \rangle \tag{1}\]

\[\mathcal{E}[x] \; \text{env} \; a = \langle \text{env}[x \leftarrow (\text{env}(x) \cup a)], \text{env}(x) \cup \{a\} \rangle \tag{\text{4}}\]

\[\mathcal{E}[e_1 \text{ op } e_2] \; \text{env} \; a = \left\{ \begin{array}{l}
\text{env} \subseteq \text{env}' \\
\text{env}', v_1 = \mathcal{E}[e_1] \; \text{env}' \; v_1 \\
\text{env}', v_2 = \mathcal{E}[e_2] \; \text{env}' \; v_2 \\
\tau = (v_1 \text{ op } v_2) \cup a
\end{array} \right\}
\text{in } (\text{env}', \tau) \tag{3}\]

\[\mathcal{E}[\text{array}(e)] \; \text{env} \; a = \left\{ \begin{array}{l}
\text{env}' \; n = \mathcal{E}[e] \; \text{env}' \; n \\
\tau = \text{Array}(n) \cup a
\end{array} \right\}
\text{in } (\text{env}', \tau) \tag{4}\]

\[\mathcal{E}[L_1, L_2, \ldots, L_n] \; \text{env} \; a = \left\{ \begin{array}{l}
\text{env} \subseteq \text{env}' \\
\text{env}'[L_1] = \tau[1] \\
\vdots \\
\text{env}'[L_n] = \tau[n]
\end{array} \right\}
\text{in } (\text{env}', \tau) \tag{5}\]

\[\mathcal{E}[\text{cond}(e_1, e_2, e_3)] \; \text{env} \; a = \left\{ \begin{array}{l}
\text{env} \subseteq \text{env}' \\
\text{env}', b = \mathcal{E}[e_1] \; \text{env}' \; b \\
\text{in}
\text{if } b \text{ then } \mathcal{E}[e_2] \; \text{env}' \; a \\
\text{else } \mathcal{E}[e_3] \; \text{env}' \; a
\end{array} \right\} \tag{6}\]

\[\mathcal{E}[e_1[e_2]] \; \text{env} \; a = \left\{ \begin{array}{l}
\text{env} \subseteq \text{env}' \\
\text{env}' \; v_1 = \mathcal{E}[e_1] \; \text{env}' \; v_1 \\
\text{env}', v_2 = \mathcal{E}[e_2] \; \text{env}' \; v_2 \\
v_1[v_2] = \tau
\end{array} \right\}
\text{in } (\text{env}', \tau) \tag{7}\]

\[\mathcal{E}[F(e)] \; \text{env} \; a = \left\{ \begin{array}{l}
\text{env} \subseteq \text{env}' \\
\text{env}', v = \mathcal{E}[e] \; \text{env}' \; v \\
\text{env}'[x], \tau = \mathcal{E}[F] \; \text{env}' \; \tau
\end{array} \right\}
\text{in } (\text{env}', \tau) \tag{8}\]

Figure 3: Abstract Semantics of Cid
\[ C[x = e] env = \]
\[ \begin{cases} \text{if } r = \top \text{ then env' else env'} \\ \text{otherwise} \\ \end{cases} \]
\[ C[def_1 ; def_2] env = \]
\[ \begin{cases} \text{if } r = \top \text{ then env' else env'} \\ \text{otherwise} \\ \end{cases} \]
\[ F[F] = \mu \lambda (v, a) \]
\[ \begin{cases} \text{if } r = \bot \text{ then env' else env'} \\ \text{otherwise} \\ \end{cases} \]

Figure 4: Abstract Semantics of declarations and functions
The pair returned is formed with a special pair constructor, written \( (, ) \), that is strict with respect to \( \top \) and with respect to \( \text{env} \), the error environment, in which every identifier is bound to \( \top \). We write \( \text{les} \) for the least common solution of a set of equations. Constraints that are inequalities of the form \( a \leq z \) are expressible as equations, \( z = a \cup x \). The function \( \text{Array} \) performs allocation of new arrays. As mentioned earlier, we assume that a function may have many local variables but only one parameter and no global variables. These assumptions are made to simplify the exposition and are not fundamental.

The semantic clauses are shown in Figure 3. Clause (8) uses the auxiliary function \( \mathcal{E}_p \), which defines the meanings of functional expressions. \( \mathcal{E}_p \) looks up the definition in the functional part of the environment giving a closure operator on the function space, i.e. an element of \( (V \rightarrow V) \rightarrow (V \rightarrow V) \). This is given by the function \( \mathcal{F}[F] \). This function is defined using a least fixed point operator, written \( \mu \), on the space of closure operators. We assume that the symbol \( F \) is bound to \( \mathcal{F}[F] \) in all the environments used in the definition of \( \mathcal{E} \).

The semantic function \( \mathcal{C}_i \), in the definition of \( \mathcal{F} \), defines the effect of declarations. The declarations take the form of equations, \( \text{ide} = \text{exp} \), and are viewed as constraints on the value of \( \text{ide} \). The abstract semantics of declarations is given in Figure 4.

5 Full Abstraction

We sketch, very briefly, the proof of the full abstraction theorem. It is carried out in three stages. First, we show that a single reduction step preserves meaning. This is a basic soundness result for the denotational semantics. Next, we show that we can always construct a reduction sequence that attains the value specified by the denotational semantics. Finally, we define a suitable operational preorder, as in Berry, Curien and Levy [1], and establish the full abstraction theorem. For this it is important that the closure operators form an algebraic cpo.

This is the traditional scenario for a full abstraction proof [6,12]. There are, however, some differences. The 'one-step reduction preserves meaning' proof requires that all solutions are preserved by reduction; fortunately, one can show this with little difficulty. The second point is that infinite objects are present in the language. Thus we cannot say that if the denotational semantics predicts a value then that value is actually attained by a reduction sequence. What we say instead is, roughly speaking, that there is a reduction sequence to every first approximation of the predicted value. The most important difference is that \( \bot \) does not model non-termination; it models a completely unconstrained value. The presence of parallel evaluation means that we need to express the possibility of a subcomputation returning a value while other subcomputations are still in progress. It is possible that one can have a terminating computation that does not impose any non-trivial constraints. For example, \( z = z \) does not impose any constraint and terminates immediately; it denotes \( \bot \). In order to get a correspondence between \( \bot \) and non-termination one needs another denotational semantics that models 'quiescence' of dataflow computations.

In order to show that one-step reduction preserves meaning we need to associate meanings with configurations. We define a semantic function \( \mathcal{M} \), in terms of \( \mathcal{E}, \mathcal{F}, \mathcal{C} \), that assigns to configurations a closure operator over the domain \( V \times \text{ENV} \). We require that the semantic environment \( \text{env} \) and the syntactic environment \( \rho \) satisfy \( \text{Dom}(\text{env}) \cap \mathcal{F}L = \emptyset \); we call this condition \( \text{\star} \), so that there will be no conflicts occurring when arrays are allocated.

We prove that the part of the environment that is initially relevant is preserved by the one-step reduction. Some of the rewrites may cause new variables to be generated; in that case one clearly cannot hope that the environments are identical. We use the notation \( \mid_{\rho(e)} \) to mean that the resulting environment is restricted to the variables that were bound in the environment \( \rho \).

**Theorem 1** Suppose that the following rewrite is possible:

\[
< D, e, \rho, \mathcal{F}L > \rightarrow < D', e', \rho', \mathcal{F}L' >
\]

then \( \forall \text{env} \) satisfying the condition \( \text{\star} \) with respect to both \( \rho \) and \( \rho' \) and \( \forall a \in V \)

\[
\mathcal{M}(\mid_D, z, \rho, \mathcal{F}L \mid_{\text{env}}) = \mathcal{M}(\mid_D', e', \rho', \mathcal{F}L' \mid_{\text{env}}) \mid_{\rho(e)}
\]

The proof is given in the full paper. It proceeds by induction on the size of the reduction sequences.

In order to state the adequacy theorem we need to relate expressions and semantic values.

**Definition 5** We say that \( e \) covers \( v \) in \( \rho \), where \( e \) is an expression, \( \rho \) is a syntactic environment, and \( v \) is a value from the semantic domain, if either (i) \( e \) is a basic value and there is a reduction sequence

\[
\emptyset, e, \rho, \mathcal{F}L \rightarrow \cdots \rightarrow \emptyset, v, \rho, \mathcal{F}L
\]

or \( v \) is an array and for all index sequences \( s \), such that \( v \cdot s \) is a basic value, we have a reduction sequence

\[
\emptyset, v \cdot s, \rho, \mathcal{F}L \rightarrow \cdots \rightarrow \emptyset, v \cdot s, \rho, \mathcal{F}L
\]
This is extended to syntactic and semantic environments in the obvious way. Intuitively, e covers v in p if we can evaluate e in p to get v without having to resolve any new constraints. The adequacy theorem can now be stated as follows.

**Theorem 2** If \( E[e] \) \( \downarrow (env',v) \) and \( p \) covers \( env' \), then:

1. if \( env' = env \) or \( v = T \) then there is a reduction of \( (\Phi, e, p, FL) \) to ERROR;
2. let \( env_f \) and \( v_f \) be finite approximations of \( env' \) and \( v \) respectively. Then there is a reduction sequence

\[
< \Phi, e, p, FL > \quad \Rightarrow \quad < D', e', p', FL' >
\]

such that \( e' \) covers \( v_f \) in \( p' \), and \( p' \) covers \( env_f \), or there is a reduction sequence to ERROR.

We omit the proof in this paper. It is quite complicated and also interesting since we have to show how we can construct the appropriate reduction sequences. We define an inclusive predicate in terms of the 'covers' relation and use a structural induction argument. We first have to prove several purely operational facts; the most important being that the rewriting is confluent. The key idea that makes the structural induction work is the following. Suppose that \( f \) and \( g \) are two closure operators that correspond to the imposition of two constraints. Suppose that we know how to construct reduction sequences corresponding to the resolution of these constraints operationally. Then, because we know that the least common fixed-point of \( f \) and \( g \) is the least fixed-point of \( f \circ g \), we can construct an interleaved reduction sequence that corresponds to computing the iterates of \( f \circ g \). In other words, the special form of the fixed-point iteration provides guidance about how one can construct the interleaved reduction sequence.

The full abstraction theorem is stated in terms of an operational preorder that relates the behaviour of expressions when they are inserted into all possible contexts.

**Definition 6** We say that \( e \sqsubseteq_{op} e' \) if in all definition contexts \( D[\cdot] \) and all basic expression contexts \( C[\cdot] \) we have that whenever there exists a reduction sequence

\[
< D[e], C[e], p, FL > \quad \Rightarrow \quad < D', b, p', FL' >
\]

then there must also be a corresponding reduction sequence of \( D[e'], C[e'], p, FL \) to \( b \) or to ERROR. If the configuration with \( e \) reduces to ERROR then that with \( e' \) must also reduce to ERROR.

This definition captures the idea that \( e \) computes to a less defined value than does \( e' \) in all contexts.

**Theorem 3**

\[
E[e_1] \quad env \ a \sqsubseteq E[e_2] \quad env \ a \quad if \quad e_1 \sqsubseteq_{op} e_2
\]

where \( env \) and \( a \) are arbitrary.

Given the adequacy result, this is relatively straightforward. Given \( e_1 \) and \( e_2 \) such that \( E[e_1] \) \( env \ a \sqsubseteq \{ E[e_2] \) \( env \ a \) we know, by algebraicity, that there is a finite element in the space of closure operators \( p \) that is less than one but not the other. We can use this to construct definition and expression contexts that distinguish \( e_1 \) and \( e_2 \).

**6 Summary and conclusions**

We have given formal operational and denotational semantics for a first-order functional language with logic variables and shown that the denotational semantics is fully abstract with respect to the operational semantics.
References


