Abstract Semantics for a Higher-Order Functional Language with Logic Variables

Radha Jagadeesan
Imperial College,
London, UK SW7 2BZ.

Keshav Pingali
Cornell University,
Ithaca, NY 14853.

Abstract

Although there is considerable experience in using languages that combine the functional and logic programming paradigms, the problem of providing an adequate semantic foundation for such languages has remained open. In an earlier paper, we solved this problem for *first-order* languages by reducing the problem to that of solving simultaneous fixpoint equations involving closure operators over a Scott domain and showing that the resulting semantics was fully abstract with respect to the operational semantics [3]. These results showed that the first-order fragment could be viewed as a language of incremental definition of data structures through constraint intersection. The problem for higher-order languages remained open, in part because higher-order functions can interact with logic variables in complicated ways to give rise to behavior reminiscent of own variables in Algol-60. We solve this problem in this paper. We show that in the presence of logic variables, higher-order functions may be modeled extensionally as closure operators on *function graphs* ordered in a way reminiscent of the ordering on extensible records in studies of inheritance [1]. We then extend the equation solving semantics of the first-order subset to the full language, and prove the usual soundness and adequacy theorems for this semantics. These results show that a higher-order functional language with logic variables can be viewed as a language of incremental definition of functions.

1 Introduction

The benefits of combining the functional and logic programming paradigms are manifold; for example, the programmer gets the power of incremental definition of data structures, which goes a long way towards solving the copy overhead of pure functional data structure construction [6, 8, 13, 11, 2]. However, it has proved difficult to find a suitable semantic foundation for such hybrid languages, which is ironic since pure functional and logic programs can be given simple abstract semantics as functions and relations over values.

In previous work, we had provided such a foundation for the first-order case by reducing the problem to that of solving simultaneous fixpoint equations involving *closure operators* over a Scott domain [3]. Using this device, we were able to provide a denotational semantics that is fully abstract with respect to the operational one. These results showed that a first-order functional language with logic variables can be viewed as a language in which data structures are defined through *constraint intersection*[1]. For a number of reasons, the problem of giving such a semantics to a *higher-order* functional language with logic variables seemed intractable for a number of reasons. As we show in Section 2, higher-order functions can interact with logic variables in very complicated ways to give rise to behavior reminiscent of own variables in Algol-60. Furthermore, these languages are inherently parallel in the sense that any correct interpreter must either be parallel or must simulate parallelism. Logic variable instantiation is like a globally visible side-effect and modeling the combination of concurrency and side-effects usually requires complex notions like powerdomains. Inspite of these apparent difficulties, we show here that in the presence of logic variables, higher-order functions may be modeled extensionally as closure operators on *function graphs* with an ordering reminiscent of the ordering on extensible records in studies of inheritance [1]. Using this tool, we are able to construct a pleasing equation solving semantics for these languages and prove the usual soundness and adequacy theorems. Our results extend the equation-solving paradigm that underlies Kahn semantics for dataflow networks [5] to a more expressive setting with higher order constructs and shared memory; this allows the communication abilities of processes to change dynamically, unlike the Kahn model of dataflow in which the channel structure of networks is fixed and

This research was performed at Cornell University under an NSF Presidential Young Investigator award (NSF grant CCR–8958543), NSF grant CCR-9008526, and a grant from the Hewlett-Packard Corporation. Correspondence regarding this paper should be sent to pingali@cs.cornell.edu.

[1] Although we did not consider non-determinism, it has been shown recently that our results extend to a first-order language with committed choice non-determinism [12].

cannot be altered during runtime.

The rest of the paper is organized as follows. In Section 2, we discuss two programs that serve to introduce the main issues and shed light on some of the difficulties in giving an abstract semantics for a functional language with logic variables. These programs are written in Id, a dataflow language that will serve as a concrete language in this paper. Section 3 gives a formal state transition semantics for Id programs. The abstract semantics is defined in Section 4. The correspondence between the operational and denotational semantics is described in Section 5. For lack of space, we omit proofs and detailed discussions from this paper and refer the interested reader to a companion technical report for details [4].

2 Informal Introduction to the Language

This section introduces Id [8] and its operational semantics informally through a number of programming examples. The core of the language is functional and logic variables are introduced through an array construct [?]. 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 program:

```
{A = array(10);
 A[1] = 2;
 fill-even(A,5);
 fill-odd(A,4);
 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
                       X[2*i+1] = X[2*i]*2 od}
```

When executed on a dataflow simulator, this program produces an array of length 10 in which the i'th element is 2^i. 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. Notice that this program cannot be executed 'sequentially' (that is, like a PASCAL or FORTRAN program); instead, computations in the calls to `fill-even` and `fill-odd` must be interleaved. Fortunately, the viewpoint of constraints provides a nice way to mask this operational complexity. For example, the definition `A = array(10)` can be viewed as a constraint that is satisfied by any array `A` of size 10 (and by an overdefined element, $\top$, which trivially satisfies all constraints). We can think of `fill-even` and `fill-odd` as constraining the even and odd elements of the array `A`, with `A` being produced by the intersection of these constraints with the constraints `A = array(10)` and `A[1] = 2`. The denotational semantics formalizes this viewpoint of constraints.

Higher order functions and logic variables

This example illustrates the interaction between higher-order functions and logic variables. Consider the program:

```
{A = array(2);
 g = f A;
 t1 = g 1;
 t2 = g 2;
 in A}

def f X i = {X[i] = i in 0}
```

The result of this program is the array [1,2]. In this program, `f` is a curried function which takes its arguments one at a time; the first argument must be an array and the second, an integer. When this function is applied to an array, it returns a 'function' that can be applied to an integer; if this new function is applied to the integer `i`, element `i` of the array gets updated to `i`. In other words, `g`, the result of applying `f` to `A`, has the array `A` embedded inside it, and this array gets updated each time `g` is applied. This is reminiscent of the behavior of *own variables* in a language like Algol-60. Furthermore, the applications of `g` need not be in the same scope as its introduction: we can pass `g` to another function and apply it inside that function.

Notice also that the right hand side of the definition of `A` (that is, `array(2)`) cannot be substituted for `A` everywhere in the program without altering the meaning of the program. Unlike in pure functional languages, *object identity* is important; in the operational semantics of Section 3, the definition of `A` will be allowed to take part in constraint solving only after the right hand side has been reduced to an array of two logic variables of the form [L1,L2].

Syntax

For the purpose of this paper, we define a core language whose syntax is shown in Figure 1. To avoid getting overwhelmed by subscripts and ellipsis, we have made this language very simple while retaining all essential

program	::=	exp
def-list	::=	def | def;def-list
def	::=	id = exp
exp	::=	const | id | exp1 op exp2
		| if exp1 then exp2 else exp3
		| array(exp) | exp1[exp2]
		| exp1 exp2 | (λx. exp)
		| def-list in exp

Figure 1: Syntax of Id

constructs. The main differences between Id , as presented earlier in the examples, and the core language are as follows. The loop construct is eliminated since a loop can be replaced by a tail recursive function. To simplify notation, we will require that all functions return a result. It is convenient to assume that the left-hand side of a definition is an identifier; a definition in Id of the form `e1[e2] = e3` can be replaced by two definitions `x = e1[e2]; x = e3` where `x` is a new identifier. We will assume that all local variables have been made into parameters so that the body of a function does not introduce any new names. We assume that the language is simply typed, and that the expressions are typed correctly in the usual sense. Arrays, booleans and integers are considered to be of base type. For definitions of the form `x= e`, `x` must have the same type as `e`. In the rest of this paper, we will ignore the details of typing. Since we do not perform unification of λ-abstractions, we impose syntactic restrictions to ensure that there are no multiple definitions of functions: if x in the abstraction $\lambda x.exp$ is of higher-order type, then x cannot occur by itself on the left hand side or the right hand side of a definition. We refer the interested reader to the companion technical report [4] for details.

3 Operational Semantics of Id

In this section, we give an operational semantics for Id using Plotkin-style [10] state transition rules. The state of the computation is represented by a *configuration* where a configuration is a quintuple $< D, e, \rho_F, \rho, FL >$. D contains definitions whose right-hand sides have not yet been completely reduced to an identifier, constant, array, or an abstraction of the form $\lambda x.exp$. The expression e in the configuration is the expression whose value is to be produced as the result 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 completely, the definition can participate in constraint solving. Configurations have two components named ρ_F and ρ which keep track of such definitions. When the right hand side of a definition in D reduces to a λ-abstraction, it is moved into ρ_F, the *function environment.* Since λ-abstractions are not unified, an identifier bound to a λ-abstraction by a definition cannot occur on the left hand side of any other definition; hence, ρ_F is simply a list of identifier/λ-abstraction pairs. The second component, ρ, called the environment, keeps track of bindings between identifiers and *base values* (identifiers, constants and arrays) and has a more complex structure to permit unification — it consists of a (possibly empty) set of *alias-sets* where an alias-set is an equivalence class of base values. For example, $\{x, y, z\}$, $\{x, y, 4\}$ and $\{x, y, [L1, L2]\}$ are alias-sets. If unification fails, the configuration is rewritten to 'Error' and computation aborts.

The transition rules for configurations are specified in terms of a binary relation $\rightarrow$ on the set of configurations. In any program P, let exp_P be the expression to be evaluated. The initial configuration for program P is $< \phi, exp_P, \phi, \phi, Id >$. We define some syntactic categories required for the operational semantics. The notation $[x_1, ..., x_n]$ for arrays represents a sequence of one or more identifiers.

$C\epsilon$ Configurations ::= $< D, e, \rho_F, \rho, FL > | \;\; Error$
$D\epsilon$ Defs ::= $\phi|\mathrm{def}_1, ..., \mathrm{def}_n$
$e\epsilon$ expression
$\rho_F\epsilon$Function_env ::= $\phi|\{f_1 = \lambda x_1.e_1, ..., f_n = \lambda x_n.e_n\}$
$\rho\epsilon$ Environment ::= $\phi|\{A_1, ..., A_n\}$
$A\epsilon$ Alias-set ::= $\{B_1, ..., B_n\}$ $B\epsilon$ Base-value ::= $x|c|Ar$
$x, L\epsilon$ Id = set of identifiers $Ar\epsilon$ Array ::= $[x_1, ..., x_n]$
$FL\epsilon$ Free-list = $\mathcal{P}$(Id)

The unification algorithm we use is similar to the one in Qute [13]. No occurs-check is performed; infinite data structures are considered to be legitimate objects of computation. The unification algorithm is defined in terms of a binary relation $\rightsquigarrow$ on environments.

Definition 1 $\rightsquigarrow$ *is a binary relation on environments defined as follows:*

1. *If A1 and A2 are members of an environment ρ, and A1 and A2 have an identifier in common, then $\rho \rightsquigarrow (\rho$ - $\{A1\}$ - $\{A2\}) \cup \{A1 \cup A2\}$.*
2. *If $\{[x_1, ..., x_n], [y_1, ..., y_n]\} \subseteq A\epsilon\rho$ then $\rho \rightsquigarrow \rho \cup \{\{x_1, y_1\}, ..., \{x_n, y_n\}\}$.*

Intuitively, these transformations leave the meaning of an environment unchanged. If $\rho_1 \rightsquigarrow \rho_2$ and $\rho_1 \neq \rho_2$, then ρ_1 is said to be *reducible*; otherwise, it is *irreducible.* Let $\overset{*}{\rightsquigarrow}$be the reflexive and transitive closure of $\rightsquigarrow$. It can be shown that for every environment ρ, there is a unique, irreducible environment ρ_1 such that $\rho\overset{*}{\rightsquigarrow}\rho_1$ [13]. If ρ is a syntactic environment and A is an alias-set, let $\mathcal{U}(\rho, A)$ denote the unique, irreducible environment such that $(\rho \cup \{A\})\overset{*}{\rightsquigarrow}\mathcal{U}(\rho, A)$.

We will need an operation that is similar to environment look-up in functional languages. In a functional language, an environment is considered to be a function from identifiers to values. In our system, the function environment ρ_F can be interpreted the same way. The rewrite rules have been designed so that in any configuration that is not Error, the environment ρ is irreducible. This means that every identifier that is not in the free-list is an element of exactly one alias-set.

Definition 2 *Let $< D, e, \rho_F, \rho, FL >$ be a configuration and x is an identifier not a member of FL. Let ρ be consistent. The function $\mathcal{V}(x)$ is defined by cases on the type of x:*

1. *x is a variable of base type: Let A be the (unique) alias-set that contains x. $\mathcal{V}(x)$ is defined by cases depending on A:*
   - *All the elements of A are identifiers. In this case, $\mathcal{V}(x)$ is undefined.*
   - *At least one element of A is a constant c. The elements of A are either identifiers or the constant c. We define $\mathcal{V}(x)$ to be c.*
   - *At least one element of A is an array. The elements of A are either identifiers or arrays of the same length. $\mathcal{V}(x)$ could be defined to be any one of these arrays. To be precise, place a lexicographical ordering on identifiers and let $\mathcal{V}(x)$ be the array whose first element is the least in this ordering.*
2. *x is a variable of a function type: In this case, $\mathcal{V}(x)$ is L where $x = L$ is the unique definition of x in ρ_F.*

The operational semantics for Id is given in Figures 4 and 5. The first rule replaces free occurrences of a first order variable x by $\mathcal{V}$(x) in any context, if $\mathcal{V}$(x) is defined. Arbitrary contexts are denoted by $C[]$ in this rule. Most of the other clauses in this semantics are self-explanatory. The two sides of a conditional expression play no role in the computation until the predicate has been evaluated to true or false. Unlike in functional languages, function application cannot be implemented by a copy of the body of the function in which occurrences of the formal parameter are substituted by copies of the actual parameter. Instead, a definition is created for the actual parameter and the actual parameter is substituted for the formal parameter only when it has been completely reduced to a base value or function.

4 Abstract Semantics

This section describes the abstract semantics for Id. First, we give an informal overview of our approach. We discuss the first-order semantics which constructs data structures through constraint intersection and we relate computing with constraints to the solution of systems of simultaneous equations involving closure operators. This part of the paper is a summary of results reported in an earlier paper [3]. Then, we show how the higher-order case fits into this picture. Next, we give a formal account of the construction of various domains needed for the formal semantic account. Finally, we present the formal semantics.

4.1 Informal Introduction

4.1.1 First-order Language

Consider the following Id program:

```
{A = array(3);
 A[1] = 2;
 A[2] = 1;
 A[3] = 3;
 in A}
```

The definition `A = array(3)` is viewed as a constraint that gives partial information about `A` - any array of length 3 this constraint. Similarly, `A[1] = 2` is a constraint satisfied by any array whose first element is 2.

How should we describe equational constraints formally? The usual powerdomain constructions are of no help here. For example, the Smythe powerdomain [15], consisting of upward closed sets, is designed to describe sets of values satisfying constraints of the form $x \sqsubseteq a$. The set of values in a domain satisfying an equational constraint is not, in general, an element of the Smythe powerdomain. Consider the constraint $x = y$. What sets of pairs satisfy this constraint? Certainly not an upward closed set because, for example, $\langle\perp,\perp\rangle$ satisfies the constraint but $\langle 2,\perp\rangle$ does not satisfy the constraint.

To motivate the formal model of constraints, note that the basic mechanism by which constraints get imposed in Id is through unification. Each time unification is performed, new constraints are imposed on some variables and this adds to the "information content" of the variables. Such functions are obviously extensive functions. Clearly, imposing a constraint twice is no different from imposing it once. Therefore, functions modeling imposition of constraints should be idempotent. Finally, we want the functions to be monotonic and continuous since the process of generating constraints is supposed to be computable.

First, we formalize the notion of "information content". If B is the domain of elementary values such as integers and booleans, consider the domain of both basic values and arrays, which can be described informally by the domain equation:

$$W = B + W + W \times W + W \times W \times W + \ldots$$

In the infinite sum, the component B represents elementary values, the component W represents arrays of length 1, the component $W \times W$ represents arrays of length 2, etc. Notice that array elements come from the domain W itself; therefore, array elements can be arrays themselves, and the domain includes 'infinitely nested' arrays. To this domain, we add an element labeled $\top$ which is a special value that models error, the result of (contradictory) definitions. A pictorial representation of the resulting domain, which we call V, is shown in Figure 2. Arrays of different lengths are incomparable. If a1 and a2 are two arrays of the same length, we say that a1 $\sqsubseteq$ a2 if a2 can be obtained by replacing occurrences of $\perp$ in a1 by other values from W. For example, the least defined array of length 3 is $[\perp, \perp, \perp]$ and it is below $[2, \perp, \perp]$ etc. The error element $\top$ is above all other values in V. This domain is constructed formally in the companion technical report [4].

We can describe the models of constraints as closure operators [14].

Definition 3 *A* closure operator*, f, on a domain V is a continuous function satisfying, (i) $\forall x \in V.\ \ x \sqsubseteq f(x)$, (ii) $f \circ f = f$.*

As an example, consider the definition `x = array(3)`. The elements of V that satisfy the constraint on x are easily seen to be solutions of the equation $x = (\lambda u.u \sqcup [\perp, \perp, \perp])x$. Note that $\lambda u.u \sqcup [\perp, \perp, \perp]$ is a closure operator. A pictorial representation of this function is shown in Figure 2 — it maps $\perp$ to $[\perp, \perp, \perp]$, the least defined array of length 3, it maps $\top$ and all arrays of length 3 to themselves, and it maps all other values in V (such as basic values and arrays of length other than 3) to $\top$.

Now that we can model constraints as closure operators, we need to understand how to model simultaneous imposition of constraints. The following lemma provides the answer.

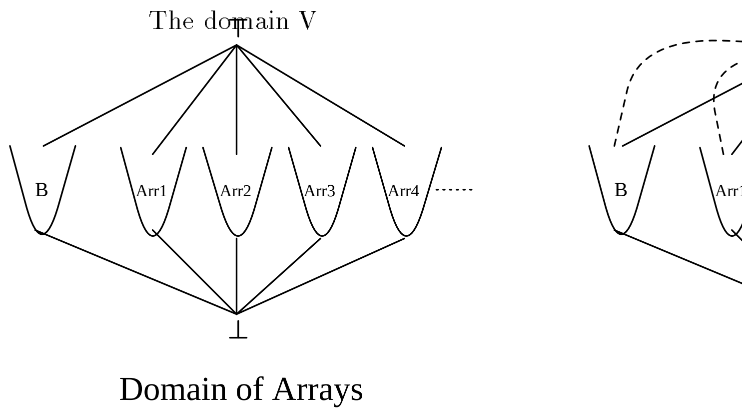
Lemma 1 *If $f : V \rightarrow V$ and $g : V \rightarrow V$ are closure operators, any solution to the system of simultaneous equations*

$x = f(x)$
$x = g(x)$

is a solution of the equation $x = f(g(x))$ and vice versa. The least common solution of the system of equations is the limit of the sequence $\perp, f(g(\perp)), f(g(f(g(\perp)))), \ldots$

This lemma lets us talk meaningfully about the least solution of a set of fixpoint equations. One interpretation of this lemma is that $\bigsqcup (f \circ g)^n$ is the smallest closure operator above f and g; hence, simultaneous imposition of constraints can be modeled using least upper bounds of closure operators.

The abstract semantics of the first-order language models definitions as closure operators on *environments* where environments are functions from identifiers to V. The interpretation of expressions is more subtle. From our previous discussion, the expression `array(3)` can be interpreted as the function $\lambda u.u \sqcup [\perp, \perp, \perp]$. Thus, `array(3)` is a closure operator of type $V \rightarrow V$. In general, we have to give meaning to an expression of the form `array(e)` where `e` can impose constraints on the environment; so, the meaning of an expression is a closure operator of type $(V \times ENV) \rightarrow (V \times ENV)$.



The closure operator for `array(3)`

Figure 2: The Domain V and a Closure Operator

4.1.2 Informal discussion of higher-order semantics

Consider the following version of the example discussed earlier in Section 2:

```
def f X i = {X[i] = i in 0}
 {A = array(2);
  g = f A;               ----(5)
  t1 = g 1;              ----(6)
  t2 = g 2;              ----(7)
  in A}
```

Function `g`, the result of applying `f` to `A`, has the array `A` "embedded" inside it, and this array gets updated each time `g` is called. The result of the program is the array $[1, 2]$.

In a pure functional language, higher-order functions are modeled by currying first-order functions. It is worth understanding why currying is inadequate for modeling the higher-order part of Id. Consider the function $F = \lambda\langle x, y\rangle.e[x, y]$ which represents a function that accepts as input a pair, say of type $D_1 \times D_2$, and returns

Figure 3: Dataflow graph for example

an element of type D_3. If v is of type D_1, the function $G = ((curry\ F)\ \ v)$ is of type $D_2 \rightarrow D_3$. This type does not model the behavior of functions in the presence of logic variables since it does not reflect the fact that v can get updated when the function G is applied, as in the example above. In a pure functional language, the value of v does not depend on what happens to G and the function G is determined entirely by F and v. This is not the case once logic variables are introduced: in our example, the value attained by array A depends on the arguments that g has been applied to.

To capture this behavior, we extend the constraint point of view developed for the first-order semantics to functions. In the higher-order semantics, function symbols like `f` and `g` are given meanings as *graphs* of input-output pairs and lambda abstractions are given meanings as closure operators on these graphs. For example, the graph of `g` will be a set of elements of the form $\langle u, v\rangle \rightarrow \langle u', v'\rangle$ where the u's and v's are integers. The intuition is that each such pair represents a piece of information about `g` : given an approximation u to the argument and v to the result, `g` refines the argument to u' and the result to v'. Function graphs get refined through application and this refinement occurs in two ways — the domain of the graph can increase or a particular element $\langle u, v\rangle \rightarrow \langle u', v'\rangle$ gets refined to $\langle u, v\rangle \rightarrow \langle u'', v''\rangle$, where $\langle u', v'\rangle \sqsubseteq \langle u'', v''\rangle$. As an example, consider Figure 3 which shows a dataflow-like representation of the example. Application nodes are made explicit as *App*, and the term λ `X.` λ `i. X[i]=i in 0` is denoted by `L`.

Initially, the graphs of `f` and `g` are $\{\ \}$ and all other variables have the value $\perp$. The two applications of `g` examine their arguments and results and add the elements $\langle 1, \perp\rangle \rightarrow \langle 1, \perp\rangle$ and $\langle 2, \perp\rangle \rightarrow \langle 2, \perp\rangle$ to graph of `g`. Also, the node `array(2)` makes its output array $[\perp, \perp]$: the array of two elements, both of which are undefined. These values are shown at the top in Figure 3.

The application node corresponding to `g = f A` collects the information about the graph of `g` and $[\perp, \perp]$ and passes it up to the node labelled `L`. Note that the use of graphs allows us to keep track of the arguments that the function has been applied to. The graph passed to `f` is $\langle [\perp, \perp], \{\langle 1, \perp\rangle \rightarrow \langle 1, \perp\rangle, \langle 2, \perp\rangle \rightarrow \langle 2, \perp\rangle\}\rangle \rightarrow$
$\langle [\perp, \perp], \{\langle 1, \perp\rangle \rightarrow \langle 1, \perp\rangle, \langle 2, \perp\rangle \rightarrow \langle 2, \perp\rangle\}\rangle$.
This is refined by the node `L` to yield the graph
$\langle [\perp, \perp], \{\langle 1, \perp\rangle \rightarrow \langle 1, \perp\rangle, \langle 2, \perp\rangle \rightarrow \langle 2, \perp\rangle\}\rangle \rightarrow$
$\langle [1, 2], \{\langle 1, \perp\rangle \rightarrow \langle 1, 0\rangle, \langle 2, \perp\rangle \rightarrow \langle 2, 0\rangle\}\rangle$
This graph is passed down to the application of `f`. This application node in turn passes down a refined version of the graph of `g`, namely $\{\langle 1, \perp\rangle \rightarrow \langle 1, 0\rangle, \langle 2, \perp\rangle \rightarrow \langle 2, 0\rangle\}$. Furthermore, it refines the value on the edge connected to the node `array(2)` to $[1, 2]$. The new value of the graph of `g` is used to update values at the application sites of `g`. For example, the application node corresponding to the statement `t2 =g 2` can now update `t2` to `0`. The graphs at this stage are shown at the bottom in Figure 3. Repeating these steps again does not alter any values. Note that the final result agrees with the answer that the operational semantics specifies.

The domain of graphs and the notion of application for graphs is specified formally in Section 4.2. As in the first-order case, definitions in the full language are interpreted as closure operators on environments. The type of expressions is also analogous to the first order case: an expression that produces a value of higher-order type (say $\sigma_1 \rightarrow \sigma_2$) will be interpreted as a closure operator on the domain $\mathcal{D}\sigma_1 \rightarrow \sigma_2 \times ENV$ where $\mathcal{D}\sigma_1 \rightarrow \sigma_2$ is the domain of graphs of type $\sigma_1 \rightarrow \sigma_2$. This domain is specified more formally next.

4.2 The Domain of Function Graphs

The domains that arise in the semantic description are complete algebraic lattices. We denote the finite elements of a domain D by $B(D)$. Given a set of ordered pairs S, define $Dom(S) = \{x \mid (\exists)\ \langle x, y\rangle \in S\}$.

Let D_1, D_2 be two domains. We first define graphs of functions from D_1 to D_2. Informally, an element of $Graphs(D_1 \rightarrow D_2)$ can be thought of as the "partial" input-output relation of a continuous function from D_1 to D_2.

Definition 4 *The set of graphs of functions from D_1 to D_2, denoted $Graphs(D_1 \rightarrow D_2)$ is defined as follows. Members of this set are sets S of elements of the form $\langle x, x'\rangle$, where $x \in B(D_1), x' \in B(D_2)$, satisfying:*

1. *Function:* $\{\langle x, x'\rangle, \langle x, x''\rangle\} \subseteq S \Rightarrow \langle x, x' \bigsqcup x''\rangle \in S$.
2. *Monotonicity:* $[\{\langle x, x'\rangle, \langle y, y'\rangle\} \subseteq S \wedge y \sqsubseteq x \wedge x' \sqsubseteq y'] \Rightarrow \langle x, y'\rangle \in S$.
3. *$Dom(S)$ is downward closed.*

The first requirement ensures that we can view graphs as encoding functions — given an element in the domain of the graph, the corresponding output is the most defined element associated with that element by the graph. Taking advantage of this, we will sometimes write $x_1 \rightarrow x_2$ when the pair $\langle x_1, x_2\rangle$ occurs in a graph. The second requirement ensures that more input guarantees more output. The final requirement clarifies the nature of the "partiality": when an element appears in the domain of the graph, all elements less than it also appear in the domain; this is justified from the operational intuition that if we apply a function to an argument, we have in effect applied it to all values less defined than the argument. Note that there are elements $S \in Graphs(D_1 \rightarrow D_2)$ such that $Dom(S)$ is not all of D_1. Thus, the elements of $Graphs(D_1 \rightarrow D_2)$ are to be distinguished from the "full" input-output relation of a continuous function from D_1 to D_2:

For the semantics, we need graphs of closure operators. The following definition picks out the graphs that correspond to closure operators by imposing the conditions of extensivity and idempotence.

Definition 5 *Let D be a domain. Then, the domain of graphs of closure operators on D, denoted $\mathcal{CG}(D)$, is defined as follows. The elements S are elements $S \in Graphs(D)$ that satisfy:*

1. *Extensivity:* $\langle x, x'\rangle \in S \Rightarrow [x \sqsubseteq x' \wedge x' \in Dom(S)]$
2. *Idempotence:* $\{\langle x, x'\rangle, \langle x', x''\rangle\} \subseteq S \Rightarrow \langle x, x''\rangle \in S$

The ordering on elements of $\mathcal{CG}(D)$ is subset inclusion.

Suppose that we are given an element d in the domain of the graph: the first condition ensures that the corresponding output e is more defined than the input element. It also ensures that the graph contains e in its domain. Now, using the second condition, we can deduce that the output predicted by the graph for e is no greater than the output predicted for d, thereby enforcing idempotence. Thus, an element of $\mathcal{CG}(D)$ can be thought of as the "partial" input-output relation of some closure operator on D. As before, the domains of elements of $\mathcal{CG}(D)$ are not required to encompass the whole of D. In this light, the ordering $S_1 \sqsubseteq S_2$ among elements of $\mathcal{CG}(D)$ implies two flavors of information: firstly, the domain of the graph S_1 is contained in the domain of S_2, and secondly, on every input in the domain of S_1 the graph S_2 yields more refined output.

Given a set S of pairs of elements from $B(D)$, let $\overline{S}$ denote the closure of S under the requirements placed on function graphs; that is, it is the smallest element of $\mathcal{CG}(D)$ containing S. If S is a singleton set $\{x\}$, we will sometimes write $\overline{x}$ instead of $\overline{\{x\}}$. It is easy to check that $\mathcal{CG}(D)$ is a complete, algebraic lattice, with the empty graph as the least element; least upper bounds given by $S_1, S_2 \in \mathcal{CG}(D) \Rightarrow S_1 \bigsqcup S_2 = \overline{S_1 \cup S_2}$; and $B(\mathcal{CG}(D)) = \{\overline{S_{fin}}\}$, where S_{fin} is any finite set of pairs of elements from $B(D)$. We can now define the domains required for the semantics. Let V be the domain of base values defined earlier. The domains at various types are defined inductively:

Base: $\mathcal{D}_o = V$.

Product spaces: $\mathcal{D}_{\sigma_1 \times \sigma_2} = \mathcal{D}_{\sigma_1} \times \mathcal{D}_{\sigma_2}$

Function spaces: $\mathcal{D}\sigma_1 \rightarrow \sigma_2 = \mathcal{CG}(\mathcal{D}_{\sigma_1} \times \mathcal{D}_{\sigma_2})$ Thus, elements of $\mathcal{D}\sigma_1 \rightarrow \sigma_2$ are sets of elements of the form $\langle x, y\rangle \rightarrow \langle x', y'\rangle$, where $x, x' \in B(\mathcal{D}_{\sigma_1})$, $y, y' \in B(\mathcal{D}_{\sigma_2})$, satisfying the requirements of Definition 5.

All of these domains are complete, algebraic lattices.

Next, we define two auxiliary functions on domains of graphs that are useful in the presentation of the semantics.

The first function is an extension of the operator that performs closure under the requirements on function graphs. Let $u \in B(\mathcal{D}_\sigma), v \in B(\mathcal{D}_\tau)$. Let $u' \in \mathcal{D}_\sigma, v' \in \mathcal{D}_\tau$ be such that $u \sqsubseteq u' and v \sqsubseteq v'$. Then, denote by $\langle u, v\rangle \hookrightarrow \langle u', v'\rangle$, the element of $\mathcal{D}\sigma \rightarrow \tau$ defined as follows:

$$\overline{\{\langle u, v\rangle \rightarrow \langle x_f, y_f\rangle \mid u \sqsubseteq x_f \sqsubseteq u', v \sqsubseteq y_f \sqsubseteq v\}}$$

The second function, App, defines the notion of application for graphs. It takes three inputs and refines them to yield three outputs: view the first coordinate as the graph of the function, the second coordinate as the graph of the argument and the third coordinate as the graph of the result. The argument and result coordinates are updated in the natural way. Furthermore, applying the graph of a function changes the graph of the function itself: it is updated to "record" the results of this application. This is exactly the behavior we required in our informal discussion in Section 4.1.2. We encourage the reader to check that the definition matches the behavior of the application node described in our informal discussion in Section 4.1.2.

Let $s \in D_{\sigma_1 \to \sigma_2}$, $t \in D_{\sigma_1}$, $u \in D_{\sigma_2}$. Then, $App(s,t,u) = (s', t', u')$, where

$$\begin{aligned} s' &= s \bigsqcup \{\langle x_f, y_f\rangle \to \langle x_f, y_f\rangle \mid x_f \sqsubseteq t \wedge y_f \sqsubseteq u\} \\ \langle t', u'\rangle &= \bigsqcup \{\langle x', y'\rangle \mid \langle x, y\rangle \to \langle x', y'\rangle \in s', x \sqsubseteq t, y \sqsubseteq u\} \end{aligned}$$

4.3 The Semantic Clauses

Figure 6 describes the denotations of definitions. The environment in which all identifiers are mapped to $\top$ is called $env_\top$. Some of the constraints are inequalities of the form $a \sqsubseteq x$, where a is a constant and x is being constrained. These can be rewritten as $x = (\lambda x. a \sqcup x)x$ in which the lambda-abstraction is obviously a closure operator. The notation **lcs** in front of a set of simultaneous equations involving closure operators stands for the *least common solution* of that set of equations. Figures 7 and 8 describe the denotations of all expressions except lambda abstraction. In the meaning of constants, the function K maps syntactic constants to their abstract equivalents. In the rule for conditionals, e_2 and e_3 play no role if e_1 is undefined. Function application is tricky since application may cause the graph of the function to change. To understand this rule write the application $e_1(e_2)$ using a prefix *Apply* operator. *App*, the closure operator that is the meaning of *Apply*, was defined in Section 4.2 and enforces constraints between e_1, e_2 and the output.

Figure 9 describes the denotation of lambda abstraction. By checking for a non-empty argument graph, the denotational semantics captures the fact that the body of a lambda expression is accessed only when it is applied to an argument. If the argument graph is non-empty, we first compute the updated environment using the function $UpdateEnv$ which essentially evaluates the body of the lambda expression in each environment obtained by binding the formal parameter to an actual parameter obtained from a, the approximation to the graph. The new environment is used to compute the new value of the graph. The case of recursion is handled implicitly by the definition of the denotation of equations. This is analogous to the handling of feedback loops by a fixpoint iteration in static determinate Kahn dataflow. The fixpoint iteration in this case is performed in the computation of the least common solution.

5 Relating the Semantic Definitions

In this section we outline the proof that the denotational semantics is correct for reasoning about the operational semantics. The interested reader is referred to a companion technical report [4] for full details. The proof extends extant proofs [3] for the first order calculus to a higher order setting.

Reduction preserves meaning

A prelude to the main adequacy result is that reduction preserves meaning. Once this is in hand, one can prove that the results obtained operationally are indeed those predicted by the denotational semantics. These proofs proceed by induction on the length of computation sequences using the basic fact that a single reduction step preserves meaning.

In order to show that one-step reduction preserves meaning we need to associate meanings with the basic entities used in the operational semantics, i.e. with configurations. The semantic function $\mathcal{M}$ assigns to configurations a closure operator over the domain $V \times ENV$. We use the semantic functions $\mathcal{E}$ and $\mathcal{C}$ defined previously. We define $\mathcal{M}[\![\langle D, e, \rho_F, \rho, FL\rangle]\!]\ \langle a, env\rangle$

$$= \ \mathbf{lcs} \left\{ \begin{array}{c} \langle a, env\rangle \sqsubseteq \langle b, env'\rangle \\ env' = \mathcal{C}[\![D \cup \rho \cup \rho_F]\!]\ env' \\ \langle b, env'\rangle = \mathcal{E}[\![e]\!]\ \langle b, env'\rangle \end{array} \right. \\ \mathbf{in}\ \langle b, env'\rangle$$

The function $\mathcal{M}$ represents the effect of the complete computation on a configuration. We prove that as we rewrite a configuration, the first order component of the result given by $\mathcal{M}$ does not alter. In particular, we show that "β"-reduction does not alter the closure operator corresponding to $\mathcal{M}$. Thus the denotational semantics "attains" the first order results predicted by the operational semantics.

The Adequacy Theorem

The hardest part of the proof of full abstraction is the converse to what is outlined in the previous subsection; namely, that every value predicted by the denotational semantics is attained by the operational semantics. Strictly speaking, we show that *for every finite approximant* to the first-order results predicted by the denotational semantics, there is a computation sequence that produces a more refined value at a finite stage.

We first define a relationship $\preceq$ between first order syntactic expressions, e, and closure operators, f, on $V \times ENV$. Intuitively, $\mathcal{E}[\![e]\!] \preceq e$ means that given any finite approximant to the result predicted by $\mathcal{E}[\![e]\!]$, there is a finite sequence of reductions evaluating e in a suitable syntactic environment, that produces a more refined value. In particular, if the result predicted by $\mathcal{E}[\![e]\!]$ is $\top$, evaluating e in a suitable syntactic environment results in *error*.

The proof that $\mathcal{E}[\![e]\!] \preceq e$, for all first order expressions e proceeds by structural induction on the expressions, and follows extant proofs [3]. The subtle case is when one has parallel imposition of constraints. We make use of the fact that the semantic prescription for determining the least common fixed point of a pair of closure operators suggests an interleaving of the reduction sequences of the subterms. More precisely, suppose that

g_1 and g_2 are two closure operators that correspond to the imposition of two constraints given as sets of equations E_1 and E_2. Suppose that we know how to construct reduction sequences corresponding to E_1 and E_2 individually. Then, since we know that the least common fixed point of g_1 and g_2 is the least fixed point of $(g_1 \circ g_2)$, we can construct an interleaved reduction sequence of E_1 and E_2 corresponding to the computing the iterates of $(g_1 \circ g_2)$. In other words, the special form of the fixed point iteration provides guidance about how to construct the interleaved reduction sequence.

The first order result is extended to the full higher order language. This is done using the idea of logical relations used in the proofs of adequacy in functional languages. Previous work [3] showed that the semantics for the first order fragment was fully abstract. We believe that with suitable restrictions on the graphs in the environment, full-abstraction for the full language can be achieved.

6 Conclusions and Related Work

We have given formal operational and denotational semantics for a higher order functional language with logic variables and shown that the denotational semantics is adequate with respect to the operational semantics.

The closest work along these lines is that of Mantha, Lindstrom and George who have given a semantics for a lazy functional language with logic variables [7]. However, this semantics encodes operational notions like suspensions and in that sense, is somewhat less abstract than our semantics which is phrased purely in terms of functions over value domains. It is possible that such operational notions are needed to model laziness, which is not required for the data-driven execution semantics of our language.

Acknowledgments: We have had stimulating discussions with Gary Lindstrom and Surya Mantha on functional languages with logic variables. We would like to thank Richard Huff for reading the paper carefully and correcting errors in the text.

References

[1] H. Ait-Kaci. *A Lattice theoritic approach to computation based on a calculus of partially ordered type structures.* PhD thesis, University of Pennsylvania, 1984.

[2] H. Ait-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

[3] R. Jagadeesan, P. Panangaden, and K. Pingali. A fully abstract semantics for a functional language with logic variables. In *Proc. of the 1989 Logic in Computer Science Conference*, 1989.

[4] R. Jagadeesan and K. Pingali. An abstract semantics for a higher-order functional language with logic variables. Technical Report TR 91-1220, Cornell University, 1991.

[5] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, pages 471–475, 1974.

[6] G. Lindstrom. Functional programming and the logical variable. In *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, 1985.

[7] S. Mantha, G. Lindstrom, and L. George. A semantic framework for functional programming with constraints. Unpublished Technical Report.

[8] R. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical Report CSG Memo 265, M.I.T. Laboratory for Computer Science, 1986.

[9] Gordon Plotkin. LCF considered a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[10] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[11] U. Reddy. *Logic Programming — Functions, Relations and Equations*, chapter On the relationship between logic and funtional languages. Prentice-Hall, 1986.

[12] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. of the conference on Principles of Programming Languages*, 1991.

[13] M. Sato and T. Sakurai. QUTE: a functional language based on unification. In *Logic Programming: functions, relations and equations*, 1986.

[14] D. Scott. Data types as lattices. *SIAM Journal of Computing*, 1976.

[15] M. B. Smythe. Powerdomains. *Journal of Computer and System Sciences*, 16:23–36, 1978.

Ident: 1. $< D, C[x], \rho_F, \rho, FL > \; \rightarrow \; < D, C[\mathcal{V}(x)/x], \rho_F, \rho, FL >$
if $\mathcal{V}(x)$ is defined

Ops: 1. $< D, e_1 \; op \; e_2, \rho_F, \rho, FL > \; \rightarrow \; < D^*, x_1 \; op \; x_2, \rho_F, \rho^*, FL^* >$
where $\{x_1, x_2\} \subseteq FL$, $FL^* = FL - \{x_1, x_2\}$, $\rho^* = \rho \cup \{\{x_1\}, \{x_2\}\}$
$D^* = D \cup \{x_1 = e_1, x_2 = e_2\}$
2. $< D, m \; op \; n, \rho_F, \rho, FL > \; \rightarrow \; < D, r, \rho_F, \rho, FL >$
if $r = m \; op \; n$

Cond: 1. $< D, cond(e_1, e_2, e_3), \rho_F, \rho, FL > \; \rightarrow \; < D^*, cond(x_1, e_2, e_3), \rho_F, \rho^*, FL^* >$
where $x_1 \in FL$, $FL^* = FL - \{x_1\}$, $\rho^* = \rho \cup \{\{x_1\}\}$, $D^* = D \cup \{x_1 = e_1\}$
2. $< D, cond(true, e_2, e_3), \rho_F, \rho, FL > \; \rightarrow \; < D, e_2, \rho_F, \rho, FL >$
3. $< D, cond(false, e_2, e_3), \rho_F, \rho, FL > \; \rightarrow \; < D, e_3, \rho_F, \rho, FL >$

Arrays: 1. $< D, array(e), \rho_F, \rho, FL > \; \rightarrow \; < D \cup \{x = e\}, array(x), \rho_F, \rho^*, FL^* >$
where $x \in FL$, $FL^* = FL - \{x\}$, $\rho^* = \rho \cup \{\{x\}\}$
2. $< D, array(n), \rho_F, \rho, FL > \; \rightarrow \; < D, [L1, ..., Ln], \rho_F, \rho^*, FL^* >$
where L1,...,Ln ϵ FL, $\rho^* = \rho \cup \{\{\text{L1}\},...\{\text{Ln}\}\}$, $FL^* = FL - \{L1, ...Ln\}$
3. $< D, e_1[e_2], \rho_F, \rho, FL > \; \rightarrow \; < D^*, x_1[x_2], \rho_F, \rho^*, FL^* >$
where $\{x_1, x_2\} \subseteq FL$, $FL^* = FL - \{x_1, x_2\}$, $\rho^* = \rho \cup \{\{x_1\}, \{x_2\}\}$
$D^* = D \cup \{x_1 = e_1, x_2 = e_2\}$
4. $< D, [L1, ..., Ln][i], \rho_F, \rho, FL > \; \rightarrow \; < D, Li, \rho_F, \rho, FL >$
where $1 \leq$ i $\leq$ n.

Function: 1.$< D, e_1(e_2), \rho_F, \rho, FL > \; \rightarrow \; < D \cup \{x_1 = e_1, x_2 = e_2\}, x_1(x_2), \rho_F, \rho, FL^* >$
where $\{x_1, x_2\} \subseteq FL$, $FL^* = FL - \{x_1, x_2\}$
2. $< D, (\lambda x.e_1)e_2, \rho_F, \rho, FL > \; \rightarrow \; < D, y = e_2 \; \textbf{in} \; e_1^*, \rho_F, \rho^*, FL^* >$
where $y \in FL$, $FL^* = FL - \{y\}$
where $e_1^* = e_1[y/x]$, $\rho^* = \rho \cup \{\{y\}\}$
3. $< D, x = e_2 \; \textbf{in} \; e_1, \rho_F, \rho, FL > \; \rightarrow \; < D^*, e1, \rho_F, \rho, FL >$
$D^* = D \cup \{x = e_2\}$

Figure 4: Structured Operational Semantics of Id: Expressions

Defs: 1. $\dfrac{< D, e, \rho_F, \rho, FL > \rightarrow < D^*, e^*, \rho_F^*, \rho^*, FL^* >}{< D \cup \{x = e\}, e_1, \rho_F, \rho, FL > \rightarrow < D^* \cup \{x = e^*\}, e_1, \rho_F^*, \rho^*, FL^* >}$
2. $< D \cup \{x = y\}, e, \rho_F, \rho, FL > \rightarrow < D, e, \rho_F, \mathcal{U}(\rho, \{x, y\}), FL >$
if x, y first order, $\mathcal{U}(\rho, \{x, y\})$ is consistent.
$< D \cup \{x = y\}, e, \rho_F, \rho, FL > \rightarrow Error$
if x, y first order, $\mathcal{U}(\rho, \{x, y\})$ inconsistent.
3. $< D \cup \{x = c\}, e, \rho_F, \rho, FL > \rightarrow < D, e, \rho_F, \mathcal{U}(\rho, \{x, c\}), FL >$
if x first order, $\mathcal{U}(\rho, \{x, c\})$ is consistent
$< D \cup \{x = c\}, e, \rho_F, \rho, FL > \rightarrow Error$
if x first order, $\mathcal{U}(\rho, \{x, c\})$ inconsistent.
4. $< D \cup \{x = [L1, ..., Ln]\}, e, \rho_F, \rho, FL > \rightarrow < D, e, \rho_F, \mathcal{U}(\rho, \{x, [L1, ..., Ln]\}), FL >$
(if $\mathcal{U}(\rho, \{x, [L1, ..., Ln]\})$ is consistent)
$< D \cup \{x = [L1, ..., Ln]\}, e, \rho_F, \rho, FL > \rightarrow Error$ (otherwise)
5. $< D \cup \{F = \lambda x.e_1\}, e, \rho_F, \rho, FL > \rightarrow < D, e, \rho_F \cup \{F = \lambda x.e_1\}, \rho, FL >$

Figure 5: Structured Operational Semantics of Id: Definitions

$$
\begin{array}{rcl}
\mathcal{C}[\![x = e]\!] \;\; env & = & \mathbf{lcs} \left\{ \begin{array}{l} env \sqsubseteq env' \\ \langle b, env' \rangle = \mathcal{E}[\![e]\!] \; \langle b, env' \rangle \\ env'[x] = b \end{array} \right. \\
 & & \mathbf{in} \; env' \\
\mathcal{C}[\![def_1 \; ; \; def_2]\!] \;\; env & = & \mathbf{lcs} \left\{ \begin{array}{l} env \sqsubseteq env' \\ env' = \mathcal{C}[\![def_1]\!] \;\; env' \\ env' = \mathcal{C}[\![def_2]\!] \;\; env' \end{array} \right. \\
 & & \mathbf{in} \; env'
\end{array}
$$

Figure 6: Denotational Semantics of Id: Definitions

$$
\begin{array}{rcl}
\mathcal{E}[\![const]\!] \; \langle a, env \rangle & = & \mathbf{lcs} \left\{ \begin{array}{l} \langle a, env \rangle \sqsubseteq \langle b, env' \rangle \\ K(const) \sqsubseteq b \end{array} \right. \\
 & & \mathbf{in} \; \langle b, env' \rangle \\
\mathcal{E}[\![x]\!] \; \langle a, env \rangle & = & \mathbf{lcs} \left\{ \begin{array}{l} \langle a, env \rangle \sqsubseteq \langle b, env' \rangle \\ env'[x] = b \end{array} \right. \\
 & & \mathbf{in} \; \langle b, env' \rangle \\
\mathcal{E}[\![cond(e_1, e_2, e_3)]\!] \; \langle a, env \rangle & = & \mathbf{lcs} \left\{ \begin{array}{l} env \sqsubseteq env' \\ \langle b, env' \rangle = \mathcal{E}[\![e_1]\!] \; \langle b, env' \rangle \end{array} \right. \\
 & & \mathbf{in} \\
 & & \mathbf{case} \; b \; \mathbf{of} \\
 & & \quad \bot \colon \langle a, env' \rangle \\
 & & \quad true : \mathcal{E}[\![e_2]\!] \; \langle a, env' \rangle \\
 & & \quad false : \mathcal{E}[\![e_3]\!] \; \langle a, env' \rangle \\
 & & \quad otherwise : < \top, env_\top > \\
 & & \mathbf{endcase} \\
\mathcal{E}[\![e_1 \; op \; e_2]\!] \; \langle a, env \rangle & = & \mathbf{lcs} \left\{ \begin{array}{l} \langle a, env \rangle \sqsubseteq \langle b, env' \rangle \\ \langle b_1, env' \rangle = \mathcal{E}[\![e_1]\!] \; \langle b_1, env' \rangle \\ \langle b_2, env' \rangle = \mathcal{E}[\![e_2]\!] \; \langle b_2, env' \rangle \\ b_1 \; op \; b_2 \sqsubseteq b \end{array} \right. \\
 & & \mathbf{in} \; \langle b, env' \rangle \\
\mathcal{E}[\![e_1(e_2)]\!] \; \langle a, env \rangle & = & \mathbf{lcs} \left\{ \begin{array}{l} \langle a, env \rangle \sqsubseteq \langle b, env' \rangle \\ (a_t, a_{arg}, b) = App(a_t, a_{arg}, b) \\ \langle a_{arg}, env' \rangle = \mathcal{E}[\![e_2]\!] \; \langle a_{arg}, env' \rangle \\ \langle a_t, env' \rangle = \mathcal{E}[\![e_1]\!] \; \langle a_t, env' \rangle \end{array} \right. \\
 & & \mathbf{in} \; \langle b, env' \rangle
\end{array}
$$

Figure 7: Denotational semantics of Id: Expressions

$$
\begin{aligned}
\mathcal{E}[\![array(e)]\!]\ \langle a, env\rangle &= \textbf{lcs}\left\{\begin{array}{l}\langle a, env\rangle \sqsubseteq \langle b, env'\rangle\\ \langle b_1, env'\rangle = \mathcal{E}[\![e]\!]\ \langle b, env'\rangle\\ \mathcal{A}rray(b_1)\sqsubseteq b\end{array}\right.\\
&\quad\ \textbf{in}\ \langle b, env'\rangle\\
\mathcal{E}[\![[L1\ldots Ln]]\!]\ \langle a, env\rangle &= \textbf{lcs}\left\{\begin{array}{l}\langle a, env\rangle \sqsubseteq \langle b, env'\rangle\\ b[i] = env'[Li]; i = 1\ldots n\end{array}\right.\\
&\quad\ \textbf{in}\ \langle b, env'\rangle\\
\mathcal{E}[\![e_1[e_2]]\!]\ \langle a, env\rangle &= \textbf{lcs}\left\{\begin{array}{l}\langle a, env\rangle \sqsubseteq \langle b, env'\rangle\\ \langle b_1, env'\rangle = \mathcal{E}[\![e_1]\!]\ \langle b_1, env'\rangle\\ \langle b_2, env'\rangle = \mathcal{E}[\![e_2]\!]\ \langle b_2, env'\rangle\\ b_1[b_2] = b\end{array}\right.\\
&\quad\ \textbf{in}\ \langle b, env'\rangle
\end{aligned}
$$

Figure 8: Denotational Semantics of Id: Array expressions

$$
\begin{aligned}
\mathcal{E}[\![\lambda x.e]\!]\ \langle a, env\rangle &= \textbf{if}\ (a = \emptyset)\\
&\quad\ \ \textbf{then}\ \langle \emptyset, env\rangle\\
&\quad\ \ \textbf{else let}\ env' = UpdateEnv^e(a)(env)\\
&\quad\quad\quad\ \textbf{in}\ \langle UpdateGraph^e(env')(a), env'\rangle\\
UpdateEnv^e(\overline{\langle u, v\rangle \to \langle u', v'\rangle}) &= \lambda\ env.\\
&\quad\ \textbf{lcs}\left\{\begin{array}{l}env[x \mapsto \overline{u'}] \sqsubseteq env'\\ \overline{v'}\sqsubseteq b\\ \langle b, env'\rangle = \mathcal{E}[\![e]\!]\ \langle b, env'\rangle\end{array}\right.\\
&\quad\ \textbf{in}\ env'[x \mapsto env[x]]\\
UpdateEnv^e(\overline{\{g_1\ldots g_n\}}) &= \lambda\ env.\\
&\quad\ \textbf{lcs}\left\{\begin{array}{l}env\sqsubseteq env'\\ env' = UpdateEnv^e(\overline{g_i}),\ i = 1\ldots n\end{array}\right.\\
&\quad\ \textbf{in}\ env'\\
UpdateEnv^e(S) &= \bigsqcup\{UpdateEnv^e(S_f)\mid S_f \sqsubseteq_f S\}
\end{aligned}
$$

$$
\begin{aligned}
UpdateGraph^e(env')(\overline{\langle u, v\rangle \to \langle u', v'\rangle}) &= \textbf{lcs}\left\{\begin{array}{l}env'[x \mapsto \overline{u'}]\sqsubseteq env^\star\\ \overline{v'}\sqsubseteq b_r\\ \langle b_r, env^\star\rangle = \mathcal{E}[\![e]\!]\ \langle b_r, env^\star\rangle\end{array}\right.\\
&\quad\ \textbf{in}\ (\langle u, v\rangle \hookrightarrow \langle env^\star[x], b_r\rangle)\\
UpdateGraph^e(env')(\overline{\{g_1\ldots g_n\}}|i = 1\ldots n\}) &= \textstyle\bigsqcup_i\{UpdateGraph^e(env')(\overline{g_i})\ |i = 1\ldots n\}\\
UpdateGraph^e(env')(S) &= \bigsqcup\{UpdateGraph^e(S_f)\mid S_f \sqsubseteq_f S\}
\end{aligned}
$$

Figure 9: Denotational Semantics of Id: Lambda terms