Abstract Semantics for a Higher order Functional Language with Logic Variables*

Radha Jagadeesan**
Keshav Pingali***

TR91-1220
July 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Correspondence regarding this paper should be addressed to Keshav Pingali.
**Radha Jagadeesan was supported by a fellowship from the Math Sciences Institute, Cornell.
***Keshav Pingali was supported by an NSF Presidential Young Investigator's award, by NSF grant CCR-9008526 and by grants from the Math Sciences Institute, Cornell, the Digital Equipment Corporation and the Hewlett-Packard Corporation.

Abstract Semantics for a Higher order Functional Language with Logic Variables*

Radha Jagadeesan†
Keshav Pingali‡
Computer Science Department
Cornell University

July 10, 1991

Abstract

The addition of logic variables to functional languages gives the programmer novel and powerful tools such as incremental definition of data structures through constraint intersection. A number of such 'hybrid' languages, like FGL + LV [11], Id [17] and Qute [24], have been implemented and are in active use. Pure functional and logic programming languages can be given elegant abstract semantics as functions and relations over values. The definition of such an abstract semantics for a functional language with logic variables has remained an open problem. In an earlier paper, we gave such a semantics for the special case of a first-order functional languages with logic variables by reducing the problem to that of solving simultaneous fixpoint equations involving closure operators over a Scott domain [9]. In fact, we obtained the rather strong result that the denotational semantics was fully abstract with respect to the operational semantics. However, 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. This problem is solved completely in this paper. We show that in the presence of logic variables, higher-order functions may be modeled extensionally as closure operators on *graphs* ordered 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.

*Correspondence regarding this paper should be addressed to Keshav Pingali.

†Radha Jagadeesan was supported by a fellowship from the Math Sciences Institute, Cornell.

‡Keshav Pingali was supported by an NSF Presidential Young Investigator's award, by NSF grant CCR-9008526 and by grants from the Math Sciences Institute, Cornell, the Digital Equipment Corporation and the Hewlett-Packard Corporation.

1 Introduction

The integration of functional and logic programming has been the subject of much research recently. From the viewpoint of logic programming, arithmetic and logical functions are necessary for efficient numerical computations [27]. From the viewpoint of functional programming, logic programming offers the novel computation paradigm of the *logic variable*, (*i.e.*, variables that are bound incrementally by constraint intersection). Incorporating logic variables into a functional language permits the elegant coding of constraint-based algorithms such as Milner's polymorphic type deduction algorithm and symbol-table management algorithms in compilers [11,22,24]. Our interest in this subject arises from the observation that logic variables can be used for efficient incremental construction of 'flat' data structures such as arrays. In a functional language, data structures are values and must be produced by a single applicative expression, which precludes incremental construction. In a language with logic variables, the programmer can work with data structures of logic variables which are bound through unification. This permits incremental array construction without the copy overhead of purely functional arrays [3].

These observations motivated the design of *Id Nouveau* [17]. Id Nouveau is a parallel programming language and has been implemented on a dataflow simulator. Several large scientific programs such as SIMPLE and particle-in-the-cell have been coded in this language [6]. Other proposals for combining functional languages with logic variables include Lindstrom's FGL + LV [11], Sato and Sakurai's Qute [24] and Ait-Kaci's Le Fun [2]. While these languages are mature, the problem of giving an appropriate denotational account for this class of languages has remained open. It is well-known that standard denotational techniques can be used to give semantics to pure functional and logic languages, using functions and relations on domains [28,12]. Surprisingly, an abstract semantic account of this kind for a functional language with logic variables has turned out to be much more difficult. The operational semantics of such a language involves both reduction, as in functional languages, and constraint solving through unification, as in logic languages. Unification of logic variables makes them aliases for the same object. The traditional denotational technique for handling aliasing is to use a two-level store [28], but this technique results in loss of abstraction since the manipulations of the store are exposed in the semantics. Another complication is that these languages are inherently parallel in the sense that any correct interpreter for this class of languages must either be parallel or must simulate parallelism, as we show in Section 2. Instantiation of a logic variable has the flavor of a globally-visible side-effect, and modeling global side-effects in the presence of concurrency is difficult, usually requiring the use of techniques like powerdomains [18]. Finally, languages like Id Nouveau are not referentially transparent, as we discuss in Section 2. All these features have made it difficult to define pleasing denotational models for such languages, and researchers have had to make do with operational models.

In an earlier paper, we gave a partial solution to the problem by giving a complete semantic account of *first-order* Id Nouveau [9]. The operational semantics was described formally using Plotkin-style structured operational semantics. The denotational account of the language related computations to solutions of simultaneous equations involving closure operators on Scott domains [25]. We were also able to show that the denotational semantics was fully abstract with respect to the operational semantics, which is a surprisingly strong result since the operational and denotational semantics have little similarity to each other. While the result for the first-order case was pleasing, it was not immediately clear how that result could be extended to the full language. In particular, the interaction between higher-order functions and logic variables is very subtle and gives rise to computational behavior reminiscent of own variables in languages like Algol-60 [16], as we show in Section 2. We could not see any way to model this in terms of equation solving.

In this paper, we give a complete solution to the problem of giving a semantic account of a higher-order functional language with logic variables. The semantics of the higher-order language is a smooth extension of the first-order semantics, and is couched purely in terms of solutions of simultaneous equations. We accomplish this by viewing higher order terms terms such as lambda abstractions as closure operators on domains of *graphs*. Just as the first-order language is a calculus of incremental definition of first-order data structures, the higher-order language can be viewed as a calculus of incremental definition of functions.

The rest of the paper is organized as follows. In Section 2, we discuss three programs that serve to introduce the language and shed light on some of the difficulties in giving an abstract semantics for a functional language with logic variables. To focus attention on key concepts, we define a core language called Cid in Section 3. Section 4 gives a formal state transition semantics for Cid programs. The abstract semantics is defined in Section 5. The correspondence between the operational and denotational semantics is shown in Section 6. We make some concluding remarks and discuss some open problems in Section 7.

2 Informal Introduction to the Language

This section introduces Id Nouveau and its operational semantics informally through a number of programming examples. For a complete description of Id Nouveau, we refer the reader to [17]. The operational semantics we present here is a simplified version of the formal operational semantics in Section 4. We assume that the reader is familiar with functional languages; therefore, we will begin by describing the array construct that uses logic variables. Then, we discuss an example that shows how arrays of logic variables can be defined incrementally. Unlike programs in conventional languages like FORTRAN or PASCAL, Id Nouveau programs cannot be executed sequentially, and the second example shows the need for parallelism in the operational model. The third example deals with the interaction of higher-order functions and logic variables. We show that higher-order functions and logic variables give rise to computational behavior similar to that of own variables in Algol-60 [16]. These examples will serve as an informal introduction to the ideas behind the operational and denotational semantics which we discuss formally in Sections 4 and 5 respectively.

2.1 Logical Arrays

To augment a functional language with logical arrays, we introduce three constructs for allocating, storing into and reading from arrays. An array is allocated by the expression

```
array(e)
```

where `e` is an expression that must evaluate to a positive integer. As is usual in functional languages, an array can be named via a definition; for example, the definition `A = array(5)` allocates an array of length 5 and names it `A`. When an array is allocated, its elements are undefined - in logic programming parlance, each element of the array is a logic variable which is uninstantiated. An element of an array A can be given a value by a definition of the form

```
A[i] = v
```

Intuitively, this has the effect of storing `v` into the `i`'th element of the array `A`. More precisely, 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, the execution of this definition results in the value `v` being stored in `A[i]`. Otherwise, if it contained some value `v1`, the result of unifying `v` and `v1` is stored into `A[i]`. If unification fails, the entire program is considered to be in error.

An element of an array may be selected by using the expression

```
A[i]
```

2.2 Incremental Array Definition

Our first example is a program for the inverse permutation problem: given an array `B` of length `n` containing a permutation of the integers 1..n, build a new array `A` of length `n` such that `A[B[i]] = i`. This is called an inverse permutation because the result array `A` contains a permutation of the integers 1..n, and when the operation is repeated with `A` as an argument, the original permutation is returned. It is straight-forward to write a program for this problem in our language.

```
def inverse-permute(B,n) =
                          {A = array(n);
                           for i from 1 to n do
                             A[B[i]] = i
                           od;
                           in A}
```

The loop construct should be thought of as syntactic sugar for tail recursion. To introduce the operational model, we discuss the execution of the call `inverse-permute([2,1,3], 3)` where the expression `[2,1,3]` denotes an array of three elements in which the first element is 2 etc. By making a copy of the body of the function in which actuals are substituted for formals, we get the following expression:

```
{ A = array(3);
  for i from 1 to 3 do          ----(1)
    A[[2,1,3][i]] = i
  od;
  in A}
```

The rewrite rule for the `array(n)` construct is `array(n)` → `[L1,...,Ln]` where the identifiers `L1,...,Ln` are new identifiers. Intuitively, this rule models the allocation of an array of length `n` in which each element is a distinct, uninstantiated logic variable. The for-loop can be replaced with copies of the loop body in which the identifier `i` is replaced by the integers 1 through 3. This results in the expression

```
{ A = [L1,L2,L3];
  A[[2,1,3][1]] = 1;
  A[[2,1,3][2]] = 2;             ----(2)
  A[[2,1,3][3]] = 3;
  in A}
```

The rewrite rule for array selection is `[X1,...,Xn][i]` → `Xi` provided `i` is an integer between 1 and `n`. Using this rewrite rule, our expression can be rewritten to

```
{ A = [L1,L2,L3];
  A[2] = 1;
  A[1] = 2;                      ----(3)
  A[3] = 3;
  in A}
```

Substituting for `A` and using the rewrite rule for array selection gives

```
{ A = [L1,L2,L3];
  L2 = 1;
  L1 = 2;                        -----(4)
  L3 = 3;
  in A}
```

which, after a few more steps, produces the result `[2,1,3]`.

Unlike in functional languages, the array `A` has not been produced as the result of evaluation of a single expression - instead, it has been defined incrementally by the co-operation of a number of definitions in the program. Abstractly, this process can be viewed in terms of constraint intersection. Consider, for example, expression (3). Each of the definitions in this expression can be thought of as constraints on the array `A`. The first definition is a constraint that asserts that `A` is an array of length 3. The second definition is a constraint that asserts that the second element of `A` is 1. In this way, each definition can be interpreted as a constraint on the value of `A` and the resulting value of `A` is obtained by intersecting all these constraints together.

We will use the viewpoint of constraints to give an abstract semantics for Id Nouveau programs. To relate the operational and denotational semantics, we show that the set of values satisfying a constraint corresponds to the set of fixpoints of a *closure operator* on the domain of values. Informally, a continuous function f is a closure operator if it is extensive ($f(x) \supseteq x$;that is, its output is always equal to or more defined than its input) and it is idempotent ($f(f(x)) = x$). For example, the definition `x = array(3)` is interpreted as the equation $x =$ *(*$\lambda(a).a \bigsqcup [\perp, \perp, \perp]$*)* x. The value $[\perp, \perp, \perp]$ is the least defined array of size 3, and it is easy to verify that the function $\lambda(a).a \bigsqcup [\perp, \perp, \perp]$ is a closure operator. Closure operators provide a nice, intuitive way of thinking about constraints and constraint satisfaction. Each definition can be viewed as a daemon that monitors program variables and refines them as necessary to satisfy the constraint it enforces. Initially, all program variables are undefined, and at the end of program execution, program variables have values agreed upon by all the daemons. In our example, the initial value of `A` is $\perp$. The first definition refines this value to $[\perp, \perp, \perp]$, which is, of course, the value obtained by applying the associated closure operator to $\perp$. This value in turn is further refined by the other definitions. At the end of program execution, the value of `A` is $[2, 1, 3]$ which is a value that all the daemons agree upon. Thus, the closure operator associated with a definition can be viewed as a description of the associated daemon and it is intuitively reasonable that such a daemon always adds information to program variables (extensivity) and that its actions are idempotent.

The evaluation of an Id Nouveau program involves both constraint solving (through unification) and reduction (such as replacing `2 + 3` by `5`). A definition of the form `A[e1] = e2` plays no role in constraint solving until `e1` and `e2` have been reduced to a value such an integer; in other words, this definition does not contribute to the value of `A` until `e1` and `e2` have been reduced to values. Some languages, such as CLP, have a more complex notion of constraint solving - for example, given the definitions `x = 2; x = y+1`, they would deduce that the value of `y` must be `1`. In our language, the definition `x = y+1` plays no role in constraint solving until the value of `y` has been produced by some other portion of the program. At that point, the value of `y+1` (call it `v`) is computed, and the definition `x = y+1` is rewritten to `x = v`. The two constraints on `x` are now solved by unifying `2` with `v`. Incorporating a more general notion of constraint solving into the language would complicate it enormously; moreover, we have not found any pressing need to do so in our domain of interest (scientific computing).

As is common in logic programming languages, we permit an unbound variable to be returned as the result (or part of the result) of executing a program. For example, if `A` in the inverse-permute problem was defined to be an array of length `n+1`, the `n+1`th element of `A` would not be defined. In our system, the resulting array would be a perfectly acceptable result (although its connection to inverse permutations is somewhat obscure!).

2.3 Need for a parallel operational model

The inverse permutation program is simple enough that it can be executed sequentially just like a FORTRAN or PASCAL program for solving the same problem. In general, an Id Nouveau program cannot be executed sequentially since the execution of a sub-expression may have to be suspended until a variable has been instantiated by another part of the program. The following program illustrates this.

```
    {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}
```

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. Procedure `fill-odd` fills in the odd elements of `A` in a similar way. Attempting to execute this program sequentially would lead to incorrect results since the second iteration of the loop in procedure `fill-even` needs the value of `X[3]`, but this value is produced by procedure `fill-odd` which has not yet been invoked. To produce the desired result, the interpreter must interleave the execution of procedure `fill-even` with the execution of procedure `fill-odd`. The Id Nouveau interpreter achieves this by selecting (non-deterministically) sub-expressions that can either be reduced or can take part in unification. In effect, the computation of `X[3]*2` in procedure `fill-even` is suspended until `X[3]` is instantiated to 8 as a result of executing sub-expressions in procedure `fill-odd`. This program shows that an abstract semantics for the language cannot be obtained merely by adding some notion of state to the semantics of the functional subset of the language. Fortunately, the viewpoint of constraints provides a nice way to mask the operational complexity - we can think of `fill-even` and `fill-odd` as constraining the even and odd elements of the array `A`, and of the array `A` as being produced by the intersection of these constraints with the constraints `A = array(10)` and `A[1] = 2`. We will exploit this idea in Section 4 to give an abstract denotational semantics for Id Nouveau.This semantics shows that one can think about the execution of Cid programs in terms of solving simultaneous equations rather than in terms of interleaved execution sequences.

2.4 High-order functions and logic variables

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

```
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}
```

In this program, `f` is a curried function which takes its arguments one at a time and for which the first argument must be an array of logic variables. When this function is applied to such an argument, it returns a function that can be applied to an integer; if this function is applied to the integer `i`, element `i` of the array gets updated to `i`. In our program, 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 [1,2]. Note that the applications of g need not be in the same scope as its introduction; for example, we can pass g to another function and apply it inside that function.

How do we model such programs denotationally? Since functions like g have updatable arrays embedded in them, the result of applying such a function could depend on the values of embedded arrays and these values in turn depend on the arguments the function has been applied to. This is a hint that the abstract semantics must have some way of keeping track of the arguments that a function has been applied to. We show that functions like g can be given extensional meanings as *graphs*. The graph of g describes the action of g on some input-output pairs. In particular, the domain of a graph need not cover the entire space of values. For example, the value of g initially is { }. Lambda abstractions and application are modeled as closure operators on graphs. Graphs of functions get refined through application and this refinement occurs in two ways — the domain of the graph can increase or the action on some input-output pair is refined further. As we will see in Section 5, the use of graphs provides a simple way to keep track of the arguments a function has been applied to, and provides a smooth extension of the first-order semantics to the higher-order case.

The first-order language is a calculus of incremental definition of first-order data structures, in which terms are denoted by closure operators that refine first order values. Similarly, the higher-order language can be viewed as a calculus of incremental definition of functions in which functions are given extensional meaning as graphs, and terms such as lambda abstractions and applications refine these higher order values. Thus, the graph associated with a function gets refined and its final value can be obtained as the result of constraint intersection. The use of 'extensible' graphs to model functions with embedded logic variables is reminiscent of the treatment of extensible records in studies of inheritance [1]. Unlike records in languages like PASCAL, extensible records can acquire more

program	::=	exp
def-list	::=	def | def;def-list
def	::=	id = exp
expression	::=	constant | id | exp1 op exp2 |
		if exp1 then exp2 else exp3 | array(exp) | exp1[exp2] |
		exp1 exp2 | ($\lambda x.$ exp) | def-list in exp

Figure 1: Syntax of Id

fields as the computation progresses. Therefore, the value of an extensible record gets refined either through refinement of an existing field or through acquisition of additional fields. If we view a record as a function from field names to values, the connection between extensible records and our model of functions becomes clear. Of course, the value domain of higher-order functions and logic variables is much more complicated than that of first-order extensible records but we hope that the analogy is illuminating.

3 Syntax of a core language

To focus on the essentials, we define a subset of Id Nouveau called Cid which is rich enough that any Id Nouveau program can be translated in a straight-forward manner into a Cid program. Working with Cid reduces the number of cases to be considered for the operational and denotational semantics.

Figure 1 describes the syntax of Cid. The main differences between Id Nouveau, as presented informally in Section 2, and the core language are as follows. The loop construct is eliminated since a loop can be replaced by a tail recursive function. In Id Nouveau, some functions, such as `inverse-permute` return a result, while others, such as `fill-even` are 'pure side-effect' functions that instantiate variables in their arguments but do not return any results. To simplify notation, we will require that all functions return a result (a function like `fill-even` can return a dummy value such as 0). It is convenient to assume that the left-hand side of a definition is an identifier; a definition in Id Nouveau of the form `e1[e2] = e3` can be replaced by two definitions `x = e1[e2]; x = e3` where `x` is a new identifier. This is reasonable if we think of definitions as constraints. In addition, 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.

As an example, consider the following variation of the `fill-even` function. The `new-fill-even-temp` function is passed `t` as a parameter every time it is called.

new-fill-even = new-fill-even-temp (array(1))
new-fill-even-temp = λt. λ X.λh.λi.
(t = X[2*i];

```
t = X[2*i-1]*2;
in if i+1 > h
     then 0
     else new-fill-even-temp array(1) X h (i+1))
```

We assume that the language is simply typed. Thus, for example, an abstraction λx. exp has type $\sigma \rightarrow \tau$ if the variable `x` has type σ and `exp` has type τ. Similarly, an application `exp1(exp2)` is well typed only if `exp1` has type $\sigma \rightarrow \tau$, `exp2` has type σ, for some σ and τ. In this case, the expression `exp1(exp2)` has type τ. 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 eschew the details of typing and assume that the expressions are typed correctly in the usual sense.

Since we do not perform unification of λ-abstractions, we impose a syntactic restriction 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.

4 Operational semantics of Id

In this section, we give an operational semantics for Id using Plotkin-style [20] state transition rules. Rather than rewrite expressions directly, it is convenient to work with *configurations*. A configuration is a quintuple $< D, e, \rho_F, \rho, FL >$ — intuitively, D contains definitions whose right-hand sides have not yet been completely reduced to an identifier, constant, array, or an abstraction of the form $\lambda x.e$. 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. For example, an occurrence of $2 + 3$ in D or in e can be replaced by 5 in a reduction step. 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 $b1$ 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. 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.

4.1 Syntactic Categories

We define some syntactic categories required for the operational semantics.

$C \epsilon$ Configurations $::= \; < D, e, \rho_F, \rho, FL > \mid \; Error$

$D \epsilon$ Defs $::= \phi | \text{def}_1, ..., \text{def}_n$ $\qquad$ $e \epsilon$ expression

$\rho_F \epsilon \text{Function} - \text{environment} ::= \phi | \{f_1 = \lambda x_1.e_1, ..., f_n = \lambda x_n.e_n\}$

$\rho \epsilon$ Environment $::= \phi | \{A_1, ..., A_n\}$
$A \epsilon$ Alias-set $::= \{B_1, ..., B_n\}$ $\qquad$ $B \epsilon$ Base-value $::= x|c|Ar$
$x, L \epsilon$ Id = set of identifiers $\qquad$ $c \epsilon$ Constant = set of constants
$Ar \epsilon$ Array $::= [x_1, ..., x_n]$

$FL \epsilon$ Free-list $= \mathcal{P}(\text{Id})$

The notation $[x_1, ..., x_n]$ for arrays represents a sequence of n identifiers, where n is greater than or equal to 1. The *length* of an array is the number of elements in this sequence.

4.1.1 Unification

The unification algorithm we use is similar to the one in Qute[24]. This is an algorithm for the unification problem in the domain of regular infinite trees. Hence, no occurs check is performed and infinite data structures are considered to be legitimate objects of computation. In a functional language, infinite data structures arise from the use of non-strict functions; for example, if `cons` is non-strict, the definition `y = cons(1,y)` defines `y` to be the infinite list of 1's. Similarly, in Id Nouveau, the programmer can write the set of definitions

```
x = array(2); x[1] = 1; x[2] = x;
```

Viewed as a tree, `x` is an 'infinitely nested' array. The unification algorithm we discuss in this section respects this intended meaning. The reader who is not interested in the details of the unification algorithm can omit this subsection.

Definition 1 *Two base values are said to be* inconsistent *if they are distinct constants, or if one is an array and the other is a constant, or if they are arrays of different lengths. This extends naturally to alias-sets and environments: an alias-set is said to be inconsistent if it contains two base values which are inconsistent, and an environment is inconsistent if it contains an alias-set that is inconsistent.*

The unification algorithm is described in terms of a binary relation $\rightsquigarrow$ on environments.

Definition 2 $\rightsquigarrow$ *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 are two transformations on environments that leave the meaning of an environment unchanged. The first clause says that in any environment, two alias-sets that contain the same identifier can be merged. The second clause says that if two arrays of the same length are in an alias-set, their elements must be in alias-sets as well.

If $\rho_1 \rightsquigarrow \rho_2$ and $\rho_1 \not\equiv \rho_2$, we say that ρ_1 reduces to ρ_2. In this case, ρ_1 is said to be *reducible*; otherwise, it is *irreducible*. Let $\overset{*}{\rightsquigarrow}$be the reflexive and transitive closure of $\rightsquigarrow$.

Theorem 1 *The relation $\overset{*}{\rightsquigarrow}$has the following properties [24]:*

1. *If $\rho_1 \overset{*}{\rightsquigarrow} \rho_2$ and $\rho_1 \overset{*}{\rightsquigarrow} \rho_3$ then $\rho_2 \overset{*}{\rightsquigarrow} \rho_4$ and $\rho_3 \overset{*}{\rightsquigarrow} \rho_4$ for some ρ_4.*

2. *There is no infinite sequence of distinct environments ρ_i such that $\rho_i \rightsquigarrow \rho_{i+1}$ for all i.*

3. *For any environment ρ, there is a unique, irreducible ρ_1 such that $\rho \overset{*}{\rightsquigarrow} \rho_1$.*

The first property states that reduction of environments has the Church-Rosser property. The second property states that an environment cannot be reduced indefinitely. The third property is an immediate consequence of the first two.

Recall that a configuration is a quintuple $< D, e, \rho_F, \rho, FL >$. When the right-hand side of a definition in D has been reduced to a base value, the definition can take part in constraint solving. If the definition is $x = b$, it can be viewed as an alias-set $\{x, b\}$. The alias-set is added to the environment ρ and this environment is reduced completely. The resulting environment incorporates all the constraints in ρ and in the definition.

Definition 3 *If ρ is an environment and A is an alias-set, let $\mathcal{U}(\rho, A)$ denote the unique, irreducible environment ρ_1 such that $(\rho \cup \{A\}) \overset{*}{\rightsquigarrow} \rho_1$. We will say that $\mathcal{U}(\rho, A)$ is the result of unifying ρ and A.*

4.2 Rewrite rules

The rewrite 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 $< \phi, exp_P, \phi, \phi, Id >$. In this configuration, D, the set of definitions to be reduced, is empty. In the initial configuration, the environment and the functional environment are empty and the free-list is Id, the set of all identifiers. A terminal configuration is one from which no transitions are possible.

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 but what about ρ? The rewrite rules have been designed so that in any configuration that is not Error, the environment is irreducible. This means that every identifier that is not in the free-list is an element of exactly one alias-set. This leads to the following definition.

Definition 4 *Let $< D, e, \rho_F, \rho, FL >$ be a configuration and x is an identifier not a member of FL. The function $\mathcal{V}(x)$ is defined by cases on the type of x A:*

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. Since A is consistent, 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. Since A is consistent, 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 intuition behind this definition is the following. During the rewrite process, occurrences of an identifier x will be replaced by $\mathcal{V}$(x) if $\mathcal{V}$(x) is defined. Consider the case when x is of base type. There is not much point to replacing one identifier with another; hence if all the elements in the alias-set of x are identifiers, we may as well make $\mathcal{V}$(x) undefined. If A contains one or more arrays, x could be replaced by any one these arrays, because the irreducibility of ρ guarantees that the elements of these arrays are themselves in alias-sets. We make $\mathcal{V}$(x) unique by our (fairly arbitrary) condition. When x is a variable of higher type, there is a unique definition for x in ρ_F.

The Plotkin-style operational semantics [20] for Id is given in Figures 2 and 3. 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. This rule, together with the first rule for definitions, ensures that a free occurrence of an identifier in a configuration can be replaced by the value associated with the identifier in the environment or functional environment. Most of the other clauses in this semantics are self-explanatory. Function application is somewhat subtle. When a function application $F(e)$ is carried out, the actual parameter e need not be a base value. Unlike in functional languages, the function application cannot simply be replaced by a copy of the body of the function in which occurrences of the formal parameter are substituted by copies of the actual parameter. Consider the Id Nouveau function

```
def F x   = {x[1] = 1;
             x[2] = 2;
            in x}
```

When F is passed an array, it stores 1 and 2 into the first and second components of the array. Consider the expression `F array(2)`. If `array(2)` is simply substituted for `x` in the body of the body of the function, the resulting expression is quite different from what one gets by first reducing `array(2)` to a base value and then performing the substitution. Looked at another way, our language is not referentially transparent and substitution must be defined carefully or we will get inconsistent results. We permit an occurrence of an identifier to be replaced by an expression only if the expression is a base value or λ-abstraction.

With this explanation, the rule for function application should be clear. A function application `F e` is rewritten by replacing it with $\texttt{exp}_F$ and adding the definitions in $\texttt{defs}_F$ to the definitions in `D`. The formal parameter `x` is renamed to avoid name clashes. Since the actual parameter `e` need not be a base value, a definition of the form `x = e` is added to the definitions in the configuration. For the same reason, arguments of predefined functions like arithmetic and array operators are also moved to `D` and reduced completely before they can participate in other computations.

4.3 Properties of the Rewrite Rules

It is straight-forward to prove a Church-Rosser theorem about the rewrite rules in Figures 2 and 3. The proof reduces to showing that Id has a one-step Church-Rosser property from which the desired theorem follows by pasting together diamonds as in proofs of the Church-Rosser theorem for lambda-calculus [4]. More precisely, we have the following development.

Definition 5 *$< D_1, e_1, \rho_{F_1}, \rho_1, FL_1 >$ and $< D_2, e_2, \rho_{F_2}, \rho_2, FL_2 >$ are* alpha-*equivalent if* $\exists x_1 \ldots x_n \in FL_1,; y_1 \ldots y_n \in FL_2$ *such that* FL_1 - $\{x_1 \ldots x_n\} = FL_2$ - $\{y_1 \ldots y_n\}$, *and replacing* $x_1 \ldots x_n$ *by* $y_1 \ldots y_n$ *in* D_1, e_1, ρ_1 *gives* D_2, e_2, ρ_2 *respectively.*

We assume the existence of a $\xrightarrow{\alpha}$ rule.

The following lemma says essentially that Id has a one-step Church-Rosser property. It can be viewed as saying that two enabled reductions do not interfere with each other.

Lemma 1 *If* $< D_0, e_0, \rho_{F_0}, \rho_0, FL_0 > \longrightarrow conf_1$ *and* $< D_0, e_0, \rho_{F_0}, \rho_0, FL_0 > \longrightarrow conf_2$, *then*

1. *If* $conf_1 = \text{error}$, $conf_2 \longrightarrow \text{error}$
2. *If* $conf_2 = \text{error}$, $conf_1 \longrightarrow \text{error}$
3. *Let* $conf_1 = < D_1, e_1, \rho_{F_1}, \rho_1, FL_1 >$, $conf_2 = < D_2, e_2, \rho_{F_2}, \rho_2, FL_2 >$, *and* $(FL_0 - FL_1) \bigcap (FL_0 - FL_2) = \Phi$. *Then one of the following holds:*

(a) $conf_1 \xrightarrow{\alpha} conf_2$

(b) $conf_1 \longrightarrow$ error, $conf_2 \longrightarrow$error

(c) $\exists < D_3, e_3, \rho_{F_3}, \rho_3, FL_3 >$ *such that* $< D_1, e_1, \rho_{F_1}, \rho_1, FL_1 > \longrightarrow < D_3, e_3, \rho_{F_3}, \rho_3, FL_3 >$ *and* $< D_2, e_2, \rho_{F_2}, \rho_2, FL_2 > \longrightarrow < D_3, e_3, \rho_{F_3}, \rho_3, FL_3 >$

Proof: The proof follows immediately from a case-by-case analysis of the rules in Figures 2 and 3, and is omitted. □

Since Id allows recursive functions, it is possible for Id programs to diverge. The following lemma states that a Id program can diverge only by making an unbounded number of function applications.

Lemma 2 *Let* $\rightarrow_s$ *be the subset of the relation* $\rightarrow$ *obtained by deleting the rule for function application. There is no infinite sequence of configurations* C_0, C_1, ... *such that for all* i, $C_i \rightarrow_s C_{i+1}$.

Proof: We define a weight function W for configurations and show that if $C_i \rightarrow_s C_{i+1}$, $W(C_{i+1}) < W(C_i)$. Informally, the weight of a configuration $< D, e, \rho_F, \rho, FL >$ is obtained by counting 1 for each identifier in D or e that is not inside array brackets, counting 2 for each operator symbol in D or e, and summing up over the configuration. It is straightforward to verify that if C1 $\rightarrow_s$ C2, then W(C2) < W(C1). This establishes the required result.□

It follows that a Id program can diverge only by performing an unbounded number of function applications.

5 Abstract Semantics

This section describes the abstract semantics for Id and is organised as follows. 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 [9]. 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.

5.1 Informal Introduction

We discuss the key ideas behind the semantics of the first-order language as a way of introducing some of the machinery required for the higher-order language.

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, $\rho^* = \rho \cup$ {{L1},...{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 2: Structured Operational Semantics of Id: Expressions

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

Figure 3: Structured Operational Semantics of Id: Definitions

5.1.1 First-order Language

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 following domain equation:

$W = B + W + W \times W + W \times W \times W + ...$

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 such as the one discussed in Section 4.1.1. To this domain, we add an element labeled $\top$ which is a special value that models error, the result of (contradictory) definitions such as the following:

```
x = 2;
x = 3;
```

A pictorial representation of the resulting domain, which we call V, is shown in Figure 4. Values in V are ordered the usual way. Arrays of different lengths are incomparable and 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 $[2, \perp, \perp]$ which in turn is below $[2, \perp, 3]$ etc. The error element $\top$ is above all other values in V. The array constructor is strict in $\top$ in the sense that any array with $\top$ as an element is identified with $\top$. This domain is constructed formally in Section 5.2.1.

Consider the following Id Nouveau program discussed in Section 2.

```
{A = array(3);
```

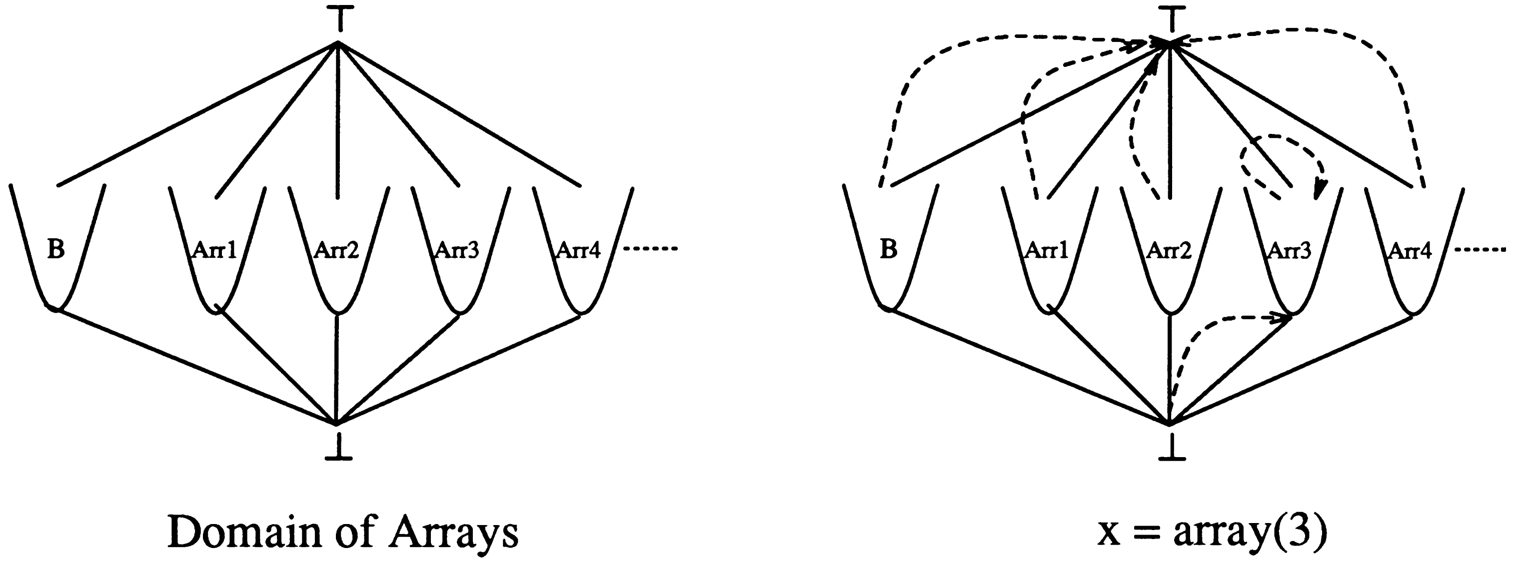


Figure 4: The Domain V and a Closure Operator on V

```
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 and the value $\top$, which satisfies all constraints trivially, satisfy this constraint. Similarly, the definition `A[1] = 2` is a constraint satisfied by any value from V that represents an array in which the first element is 2, and by $\top$. Composition of definitions is treated as simultaneous constraint imposition.

How should we describe formally the subsets of V that correspond to solutions of constraints in our language? The usual powerdomain constructions are of no help here. For example, the Smythe powerdomain [26], consisting of upward closed sets, is designed to describe sets of values satisfying constraints of the form $x \sqsubseteq a$. However, the constraints we are interested in expressing are equational constraints. 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.

The key observation that provides the foundation of the abstract semantics is that the sets of solutions to constraints can be modeled as retractions of V - that is, they can be described as solutions of fixpoint equations of the form $x = f(x)$ where $f : V \rightarrow V$ is continuous and idempotent. Moreover, the retraction mappings f are *extensive* functions ($x \sqsubseteq f(x)$). Functions that are continuous, extensive and idempotent are called closure operators [25].

As an example, consider the definition `x = array(3)`. The elements of V that satisfy the constraint are easily seen to be solutions of the equation $x = (\lambda u.u \sqcup [\perp, \perp, \perp])x$. A pictorial representation of this function is shown in Figure 4 - 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 $\top$. It is easy to verify that $(\lambda u.u \sqcup [\perp, \perp, \perp])$ is a closure operator. As another example, the closure operator corresponding to a definition of the form `x = y` is $\lambda v.[v[1] \sqcup v[2], v[1] \sqcup v[2]]$.

There is a deep connection between the operational semantics and the fact that solutions of constraints in our language can be described as fixpoints of closure operators. The basic mechanism by which constraints get imposed in Id Nouveau 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 as well since the process of generating constraints is supposed to be computable. For future reference, we write down some facts about closure operators.

Definition 6 *A* closure operator, f, *on a domain* V *is a continuous function satisfying, (i)* $\forall x \in V.x \sqsubseteq f(x)$, *(ii)* $f \circ f = f$.

Lemma 3 *The collection of closure operators* $V \rightarrow V$ *is itself a complete partial order in which the least element is the identity function.*

Now that we can model the sets of values satisfying constraints as fixpoints of closure operators, we need to understand how intersection of such sets fits into our framework. The following lemma, whose proof is elementary, provides the answer.

Lemma 4 *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 compute solutions of a set of fixpoint equations involving closure operators, and talk meaningfully about the least solution of such a set of equations[1]. In abstract terms, intersection of sets is modeled using composition of closure operators. An interesting fact is that if f and g are closure operators, then $f \circ g$ is not necessarily a closure operator, but $\bigsqcup(f \circ g)^n$ is a closure operator with the same fixpoints as $f \circ g$.

[1]Note that if we assume only that f and g are continuous functions, the system of equations in Lemma 4 need not have a solution, let alone a least solution.

An operational interpretation of this is that a value satisfying both constraints can be obtained by interleaving executions of f and g until 'steady-state' is reached - of course, this interpretation has nothing to do with the actual operational semantics but it provides the key to the proof of correspondence between the operational and denotational semantics [9].

The abstract semantics of the first-order language models definitions as closure operators on *environments* where environments are functions from identifiers to V. For example, the meaning of the definition `x = y + 3` is the closure operator g defined by $\lambda env.\ env[x \mapsto env(x) \bigsqcup (env(y) + 3)]$. Similarly, the meaning of the definition `y = 1` is the closure operator f defined by $\lambda env.\ env[y \mapsto env(y) \bigsqcup 1]$. Consider the two definitions together: `x = y+3; y = 1`. Let the meaning of the two definitions together be denoted by h. Let $env_\perp$ be the environment in which all identifiers are undefined. Then, from lemma 4, $h\ env_\perp$ is the limit of the sequence $env_\perp, f(g(env_\perp)), f(g(f(g(env_\perp)))), ...$ Note that

$$
\begin{aligned}
&g(env_\perp) = env_\perp \\
&f(g(env_\perp)) = f(env_\perp) = env_\perp[y \mapsto 1] \\
&g(f(g(env_\perp))) = g(env_\perp[y \mapsto 1]) = env_\perp[x \mapsto 4, y \mapsto 1] \\
&f(g(f(g(env_\perp)))) = g(f(g(env_\perp)))
\end{aligned}
$$

Thus, the result of evaluating the two definitions simultaneously in $env_\perp$ results in the environment $env_\perp[x \mapsto 4, y \mapsto 1]$ as expected.

The interpretation of expressions is more subtle. Looking back at our discussion of the meaning of the definition `x = array(3)`, we observe that the expression `array(3)` can be interpreted as the function $\lambda u.u \sqcup [\perp, \perp, \perp]$. Thus, `array(3)` is a function of type $V \rightarrow V$ which, given a value, refines it to a more defined value that is either $\top$ or an array of length 3. More generally, we must give meaning to an expression of the form `array(n)`; so we let the function take an additional argument, the environment, so that it can look up values of identifiers, such as `n`. In general, we have to give meaning to an expression of the form `array(e)` where `e` can impose constraints on the environment; so, we also make the function return an environment as a result. Therefore, the meaning of an expression is a closure operator of type $(V \times ENV) \rightarrow (V \times ENV)$.

The interpretation of user-defined functions follows naturally from the interpretation of expressions. In the first-order semantics, a user-defined function with one parameter is interpreted as a closure operator of type $(V \times V) \rightarrow (V \times V)$ — given values for the argument and result, the function refines them, producing new values.

5.1.2 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 an element of type D_3. Currying this function gives a function of type $D_1 \to D_2 \to D_3$. If v is of type D_1, the function $G = ((CURRY F)\ \ v)$ is of type $D_2 \to D_3$. 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 is applied, as in the example discussed 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.

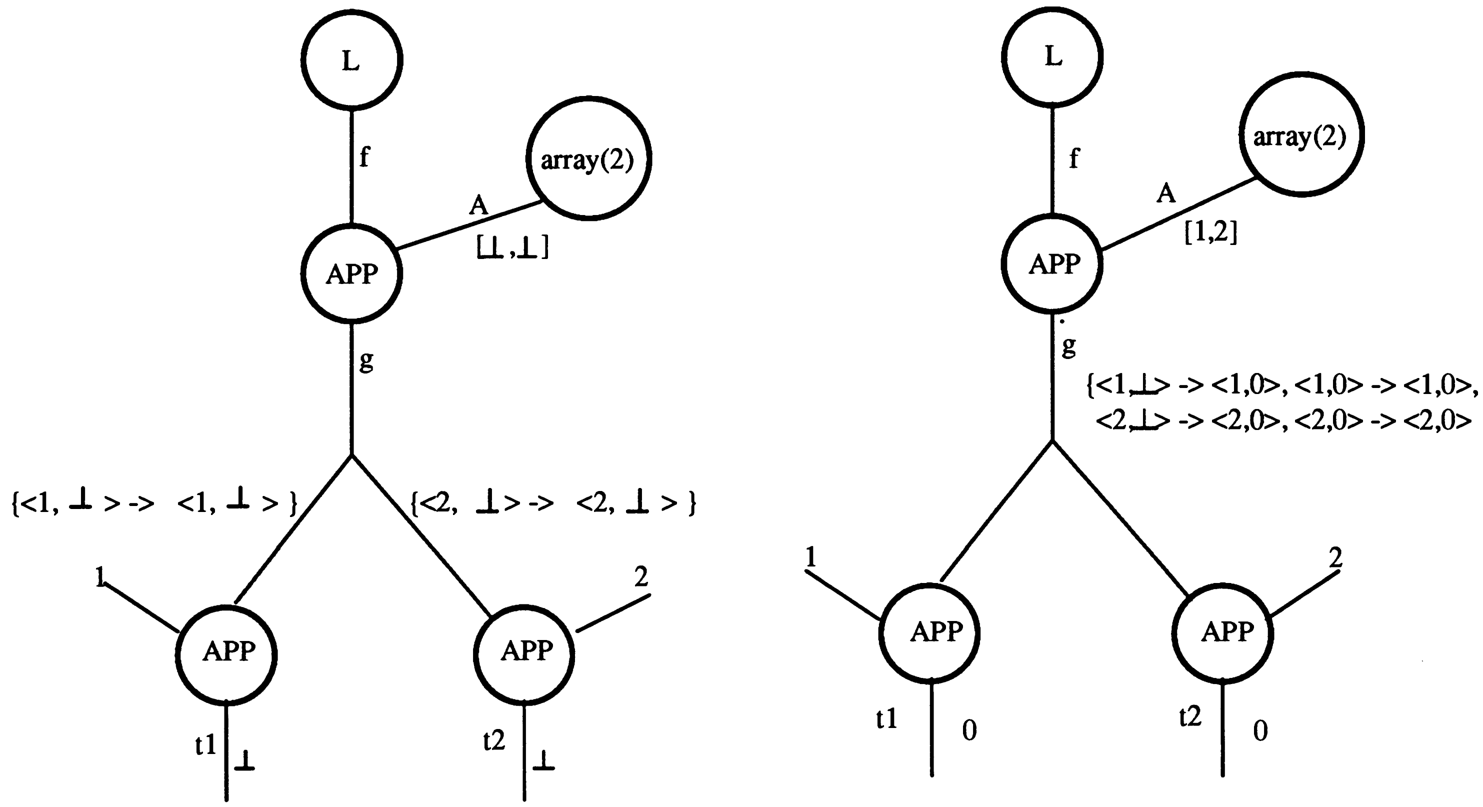
To capture this behavior, we extend the constraint point of view developed for the first-order semantics to functions. In the higher-order semantics, functions like `f` and g are given meanings as *graphs*, and lambda abstractions are given meanings as closure operators on these graphs: for example the graph of g will be a set of elements of the form $\langle u, v\rangle \to \langle u', v'\rangle$. 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 $\langle u, v\rangle \to \langle u', v'\rangle$ gets refined to $\langle u, v\rangle \to \langle u'', v''\rangle$, where $\langle u', v'\rangle \sqsubseteq \langle u'', v''\rangle$. To understand this better, consider Figure 5 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 $\langle 1, \perp\rangle \to \langle 1, \perp\rangle$ and $\langle 2, \perp\rangle \to \langle 2, \perp\rangle$ to graph of g. Also, the node `array(2)` makes the graph on its edge $[\perp, \perp]$: the array of two elements, both of which are undefined. These values are shown in the diagram on the left in Figure 5.

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 functions has been applied to. The graph passed to F is

$$\{\langle[\perp, \perp], \{\langle 1, \perp\rangle \to \langle 1, \perp\rangle, \langle 2, \perp\rangle \to \langle 2, \perp\rangle\}\rangle \to \langle[\perp, \perp], \{\langle 1, \perp\rangle \to \langle 1, \perp\rangle, \langle 2, \perp\rangle \to \langle 2, \perp\rangle\}\rangle\}$$

This is refined by the node `L`. The resulting graph is

$$\{\langle[\perp, \perp], \{\langle 1, \perp\rangle \to \langle 1, \perp\rangle, \langle 2, \perp\rangle \to \langle 2, \perp\rangle\}\rangle \to \langle[1, 2], \{\langle 1, \perp\rangle \to \langle 1, 0\rangle, \langle 2, \perp\rangle \to \langle 2, 0\rangle\}\rangle\}$$



INTERMEDIATE STAGE FINAL ANSWERS

Figure 5: Dataflow graph for example

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 $\{\langle 1,\perp\rangle \to \langle 1,0\rangle, \langle 2,\perp\rangle \to \langle 2,0\rangle\}$. 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 in the diagram on the right in Figure 5. Repeating these steps again does not alter any values. Note that the final result yielded agrees with the answer that the operational semantics predicts.

The domain of graphs and the notion of application for graphs is specified formally in Section 5.2.2. As in the first-order case, definitions in the full language are interpreted as closure operators on environments. An expression of higher-order type (say $\sigma_1 \to \sigma_2$) will be interpreted as a closure operator on the domain $G\sigma_1 \to \sigma_2 \times ENV$ where $G\sigma_1 \to \sigma_2$ is the domain of graphs of type $\sigma_1 \to \sigma_2$.

5.2 The Semantic Domain

Section 5.2.1 gives a formal definition of the domain V that was introduced informally in Section 5.1.1. Section 5.2.2 is a formal definition of the domain of graphs introduced informally in Section 5.1.2.

5.2.1 Domains for first-order language

To define the domain of arrays, we use a standard construction for defining a domain of (possibly infinite) terms in logic programming, as described in Lloyd's book [12]. We include it here for completeness. First we need some notation. Let ω be the set of natural numbers. We use ω^* for the set of finite sequences of natural numbers. 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 to the end. The size of a set X is witten $|X|$ and the size of a sequence s is written $|s|$.

Definition 7 *A* tree T *is a subset of* ω^* *satisfying*

1. $\forall s \in \omega^*$ *and* $\forall i,j \in \omega$ *we have* $([s,i] \in T \wedge j < i) \Rightarrow (s \in T \wedge [s,j] \in T)$.

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. We define $br(s,t)$ to be the number of successors of the node s in the tree t; if the tree is clear from context we will write $br(s)$. 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 we add a top element, written $\top$. The domain W is defined as follows. Let *Atom* be a given domain of atomic values and let *Arrays* be the set of array constructors written in infix form as $\{[\ ],[,],[,,],\ldots\}$ or for ease of reference as $\{array_1, array_2, \ldots,\}$. Let $A = Atom \cup \{\Omega\} \cup Arrays$ where Ω stands for the undefined element.

Definition 8 *An element of W is a function $f : t \to A$ where t is a non-empty tree. The function f satisfies $\forall s \in t.br(s) = 0 \Rightarrow f(s) \in (Atom \cup \Omega) \wedge br(s) = n \neq 0 \Rightarrow f(s) = array_n$. The ordering between elements of W is defined as follows: $f \sqsubseteq g$ iff*

- $dom(f) \subseteq dom(g)$
- $\forall s \in dom(f)$
  1. $br(s, dom(f)) \neq 0 \Rightarrow br(s, dom(g)) = br(s, dom(f))$
  2. $br(s, dom(f)) = 0 \Rightarrow f(s) = \Omega \vee g(s) = f(s)$

The ordering between elements of W allows one to replace occurrences of Ω with other elements to obtain a larger element. This domain describes infinitely deeply nested arrays but all arrays must have finite "width". Note that if two arrays have different widths they are incomparable. Thus the domain decomposes into subdomains corresponding to different array sizes, as shown in Figure 4. It is straightforward to check that the domain is algebraic and consistently complete.

5.2.2 Domains of Function Graphs

In the discussion below, D is a complete algebraic lattice. We denote the finite elements of a domain D by $B(D)$; given a set of ordered pairs S, we denote by $Dom(S)$ the domain of S; more precisely: $Dom(S) = \{x \mid (\exists)\ \langle x, y\rangle \in S\}$

Definition 1 *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 are subsets S of elements of the form $\langle x, x'\rangle$, where $x, x' \in B(D)$, satisfying the following requirements:*

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

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

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 and x_2 is the most defined element associated with x_1. The fourth requirement, together with the first, ensures idempotence. These requirements are reasonable since we are dealing with graphs of closure operators.

The final requirement is that 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.

Given a subset 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 smallest element of $\mathcal{CG}(D)$ containing S. If S is a singleton set $\{x\}$, we will sometimes write $\overline{x}$ instead of $\overline{\{x\}}$.

The following lemma establishes that $\mathcal{CG}(D)$ has the desired properties.: *i.e.*, it is a complete, algebraic lattice. The proof is straightforward and is omitted.

Lemma 5 *Let D be a complete algebraic lattice. Then, $(\mathcal{CG}(D), \subseteq)$, is a complete algebraic lattice, with*

- *Least element: the empty graph*
- $S_1, S_2 \in \mathcal{CG}(D) \Rightarrow S_1 \bigsqcup S_2 = \overline{S_1 \cup S_2}$
- $B(\mathcal{CG}(D)) = \{\overline{S_{fin}}\}$, *where S_{fin} is any finite subset of pairs of elements from $B(D)$.*

Given this construction, we can now define the domains required for the semantics. Let V be the domain of base values. Then, the domains at various types are defined inductively as follows:

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 1.

Note that the notions of argument and result of a user-defined function get fuzzy in our semantics — the interpretation of a function is that given approximations to the argument and result, the function returns refinements of these.

The following lemma establishes that the domains have the desired properties: *i.e.*, all the domains of graphs are complete, algebraic lattices. The proof proceeds by induction on types and is omitted.

Lemma 6 *For all types $\sigma_1 \rightarrow \sigma_2$, $(\mathcal{D}\sigma_1 \rightarrow \sigma_2, \subseteq)$, is a complete algebraic lattice, with*

- *Least element: the empty graph*
- $S_1, S_2 \in \mathcal{D}_{\sigma_1 \rightarrow \sigma_2} \Rightarrow S_1 \bigsqcup S_2 = \overline{S_1 \cup S_2}$
- $B(\mathcal{D}_{\sigma_1 \rightarrow \sigma_2}) = \{\overline{S_{fin}} |\ S_{fin}\}$*, where* S_{fin} *is any finite set of elements of the form,* $\langle x, y\rangle \rightarrow \langle x', y'\rangle$*, where* $x, x' \in B(\mathcal{D}_{\sigma_1})$*,* $y, y' \in \times B(\mathcal{D}_{\sigma_2})$

Next, we define some 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. The second function, *App*, defines the notion of application for graphs — note that applying a graph can change the graph itself, which is exactly the behavior we required in our informal discussion in Section 5.1.2.

1. 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 $u \sqsubseteq u', v \sqsubseteq v'$. Then, denote by $\langle u, v\rangle \hookrightarrow \langle u', v'\rangle$, the element of $\mathcal{D}_{\sigma \rightarrow \tau}$ defined as follows:
$$\overline{\{\langle u, v\rangle \rightarrow \langle x_f, y_f\rangle | u \sqsubseteq x_f \sqsubseteq u', v \sqsubseteq y_f \sqsubseteq v\}}$$
2. The following definition describes the notion of the application of a graph to an argument. Let $s \in D_{\sigma_1 \rightarrow \sigma_2}$, $t \in D_{\sigma_1}$, $u \in D_{\sigma_2}$. Then,

$$\begin{aligned}
App(s, t, u) &= (s', t', u'), where \\
s' &= s \bigsqcup \{\langle x_f, y_f\rangle \rightarrow \langle x_f, y_f\rangle |\ x_f \sqsubseteq t \wedge y_f \sqsubseteq u\} \\
\langle t', u'\rangle &= \bigsqcup \{\langle x', y'\rangle |\ \langle x, y\rangle \rightarrow \langle x', y'\rangle \in s' \wedge\ x \sqsubseteq t \wedge y \sqsubseteq u\}
\end{aligned}$$

5.3 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 appear to be inequalities rather than equalities. However, the inequalities are all of the form $a \sqsubseteq x$, where a is a constant and x is being constrained. These can be rewritten as $x = a \sqcup x$ which in turn can be written as $x = (\lambda x.a \sqcup x)x$ in which the lambda-abstraction is obviously a closure operator. A definition incorporates all the constraints in its right hand side expression and also constrains the identifier on the left hand side. Composition of definitions is treated as a simultaneous fixpoint of closure operators as discussed in Section 5.1.1. 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. The existence of such a solution follows from Lemma 4.

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, note that e_2 and e_3 play no role if e_1 is undefined. Function application is the only tricky clause since application may cause the meaning of the function to change. A simple way to make sense of this rule is to write application as *Apply*(e_1, e_2), using a prefix *Apply* operator. *App*, the closure operator that is the meaning of *Apply*, was defined in Section 5.2.2 and enforces constraints between e_1, e_2 and the output, refining the value of e_1 if necessary.

Figure 9 describes the denotation of lambda abstraction. Recall from the operational semantics that the body of a lambda expression is accessed only when it is applied to an argument. This is mirrored in the denotational semantics in the check for non-empty argument graph. Thus the constraints imposed by the body of the lambda expression are taken into account only when the argument graph is non-empty. Otherwise, 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. Notice that there is no special case for recursion. 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.

5.4 Properties of the semantics

For the proofs of correspondence of the operational and abstract semantics, it is useful to note the following property of the closure operators used in the semantics of expressions. Consider the meaning of identifiers. Notice that the closure operator satisfies the following condition:

$$\mathcal{E}[\![x]\!]\ a\ env = a\ env\ \wedge\ \mathcal{E}[\![x]\!]\ b\ env = b\ env \Rightarrow \mathcal{E}[\![x]\!]\ a \bigsqcup b\ env\ = a \bigsqcup b, env\rangle$$

For a general closure operator f, we can write this condition as follows:
$\langle a, env\rangle \in Fix(f)\ \wedge\ \langle b, env\rangle \in Fix(f) \Rightarrow \langle a \bigsqcup b, env\rangle \in Fix(f)$.

We call this condition *additivity*. It is easy to check that the parallel composition of additive closure operators is additive. Furthermore, the additive closure operators are closed under limits of directed sets. It is easy to see that all the closure operators that are denotations of expressions satisfy this condition.

6 Relating the Semantic Definitions

In this section we outline the proof that the denotational semantics is correct for reasoning about the operational semantics. Because of the length and density of these proofs, we leave the details to the appendices.

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

$$\mathcal{C}[\![def_1\ ;\ def_2]\!]\ env \;=\; \text{lcs}\left\{\begin{array}{l} env \sqsubseteq env' \\ env' = \mathcal{C}[\![def_1]\!]\ env' \\ env' = \mathcal{C}[\![def_2]\!]\ env' \end{array}\right.$$
$$\textbf{in } env'$$

Figure 6: Denotational Semantics of Id: Definitions

6.1 One step reduction preserves meaning

A prelude to the main adequacy result is that a single reduction step 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. In the following, 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 and the same notational conventions.

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

The function $\mathcal{M}$ represents the effect of the complete computation on a configuration. The theorem we will prove shows that as we rewrite a configuration the first order component of the result given by $\mathcal{M}$ will not alter. All the cases of the proof are identical to extant proofs for the first order calculus [9] except the proofs for two rules: "β"-reduction, and the rule that replaces function names by definitions. The appendix spells out these proofs in detail.

6.2 The Adequacy Theorem

The hardest part of the proof 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

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

Figure 7: Denotational semantics of Id: Expressions

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

Figure 8: Denotational Semantics of Id: Array expressions

the 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 $V \times ENV$. Previous work [9] proves that for all syntactic expressions e, $\mathcal{E}[\![e]\!] \preceq e$. 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 expressions e proceeds by structural induction on the expressions, and has been done in the paper on the semantics of the first order language [9]. The difficult part of the proof is that one has to construct a reduction sequence from semantic information. In our case, we use the special properties of fixed points of closure operators to carry out this construction. For most of the constructs in the language it is clear how one ought to proceed. The subtle case is when one has parallel imposition of constraints. The proof made 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.

In the appendix, we extend the first order result to the full higher order language. This is done using the idea of logical relations used in the proofs of adequacy in usual functional languages [19]. Thus, we define a notion of computability of terms by induction on types.

Definition 1 *(Definition of the computability predicate for closed terms)*

- *$Comp(e)$ is true, for an expression e of base type if $\mathcal{E}[\![e]\!] \preceq e$*

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

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

Figure 9: Denotational Semantics of Id: Lambda terms

- *$Comp(e)$ is true, for an expression e of type $\sigma_1 \rightarrow \sigma_2$, if $(\forall e')$, e' of type σ_1, such that $Comp(e')$, $Comp(e(e'))$*

We extend the definition to arbitrary open terms by defining the notion of a valid substitution. The main theorem shows that all terms are computable.

Theorem 2 *All terms are computable.*

6.3 Full Abstraction

Previous work [9] showed that the semantics for the first order fragment was fully abstract. This full abstraction result should be viewed in the light of previous work on full abstraction for functional languages [19,15,5]. Full abstraction for functional languages was first studied by Plotkin in his seminal paper on LCF where full abstraction was obtained by adding parallel-or construct to the language [19]. Intuitively, the parallel-or operator allowed the syntactic definition of the semantic least upper bound function. The semantics of the first order language is fully abstract without the addition of new syntactic constructs because the least upper bound function is introduced implicitly into the language through composition of definitions.

It remains an open question if the semantics that is presented here for the higher order language is fully-abstract. The subtleties arise because the semantics does not place sufficient restrictions on the values of the graphs that go into $env[F]$, where F is a function symbol. In particular, a semantic environment env may have an approximation $env[F]$ such that the graph $env[F]$ is not "realized" by the denotation of L. For example, consider the definition $F = \lambda x.x$. Consider the graph $\overline{\{\langle 1,1\rangle \rightarrow \langle 1,T\rangle\}}$. This graph has information that cannot arise in an operational reduction sequence. We believe that with suitable restrictions on the graphs in the environment, full-abstraction can be achieved.

7 Conclusions

In this paper, we gave a formal account of a higher-order functional language extended with logic variables. The operational semantics incorporated an explicit treatment of aliasing introduced by unification, and the interleaving of computations in different parts of the program. While such features usually complicate the denotational model, we showed that Cid programs could be given a simple denotational treatment couched entirely in terms of equations and equation solving. The connection between equation solving and the operational semantics was established through a novel use of closure operators on Scott domains. These results show that the first-order language can be viewed as a language in which data structures are defined incrementally through constraint intersection. Similarly, the higher-order language can be viewed as a language in which functions are defined incrementally through constraint intersection, where functions are modeled extensionally as graphs of closure operators. The ordering on functions is reminiscent of the ordering on extensible records in studies of inheritance.

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 [13]. 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.

It has been suggested that the purpose of semantics is to guide the development of appropriate syntax [7]. Therefore, it would seem to be appropriate to modify the syntax of Cid definitions so that they look more like assertions about objects than bindings of names to values. For example, the definition `x = array(5)` could be rewritten as an assertion `length(x,5)` where `length` is a binary predicate that is true if the length of the first argument is equal to the second. Composition of definitions could be written as conjunction of predicates and user-defined functions could look like user-defined predicates. Unfortunately, this relational syntax has no information about the *modes* of the arguments of predicates — should the evaluation of the predicate `length(x,5)` create an array, like the Cid definition did, or should it check the length of an existing array and return true or false? The logic programming community has explored this issue using annotations of various kinds [27], but we believe that the development of a clean syntax for this class of languages is still an open issue.

Although the results of this paper are presented in the context of Id Nouveau, the techniques we have developed can be used to give semantics to other concurrent constraint programming languages such as GHC or Janus. The case of deterministic constraint programming languages is straight-forward since they incorporate only AND-parallelism, and the techniques of this paper can be applied directly. The extension of this work to the non-deterministic setting is still in a state of flux; one possible approach is described in [23].

For researchers interested in the semantics of concurrency, we point out that the semantics in this paper extends the equation-solving paradigm that underlies Kahn semantics for dataflow networks [10] to a more expressive setting in which processes manipulate shared memory locations. Kahn's dataflow networks communicate by sending messages on channels, and message transmission is monotonic in the sense that a message cannot be deleted or recalled once it has been sent. Our framework extends this model with monotonic shared memory in which globally visible names are bound to values through unification. This extension is non-trivial because it 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 cannot be altered during runtime. The issue of channels as first class objects has also arisen in the context of process calculi. In traditional process calculi [14,8], channels cannot be passed around freely. Lately, researchers have investigated process calculi where processes can transfer communication abilities to other processes [21]. The semantics presented here presents a simple description of determinate processes that have this capability.

Acknowledgments: We thank Hassan Ait-Kaci, Gary Lindstrom, Albert Meyer and

Jean-Louis Lassez for useful conversations about functional languages and logic variables.

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] Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11, October 1989.

[4] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* Studies in Logic. North-Holland, Amsterdam, revised edition edition, 1984.

[5] G. Berry, P. L. Curien, and J. J. Levy. Full abstraction for sequential languages; the state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 3, pages 89–132. Cambridge University Press, 1985.

[6] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The Simple code. Technical Report UCID-17715, Lawrence Livermore Laboratory, 1978.

[7] J-Y. Girard. personal communication, 1987.

[8] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, London, 1985.

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

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

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

[12] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

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

[14] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[15] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–23, 1977.

[16] P. Naur. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, January 1963.

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

[18] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.

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

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

[21] J. Parrow R. Milner and D. Walker. Mobile processes. Technical report, University of Edinburgh, 1989. Draft.

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

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

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

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

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

[27] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1986.

[28] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

A One-step reduction preserves meaning

In this section we will show that the reduction relation preserves meaning, as given by the abstract semantics. This shows that if a sequence of rewrites leads to a value that cannot be reduced any further then this value is the one predicted by the abstract semantics. For this we need to translate the syntactic environment and the unresolved constraints into a set of equations. We formalize this notion first. This discussion is based on the analogous proof for the first order fragment [9].

A syntactic environment ρ is a collection of alias sets and each alias set is a set consisting, in general, of identifiers and terms. Suppose that ρ is a syntactic environment, we shall write $EQ(\rho)$ for the set of equations generated from ρ. We define $EQ(\rho)$ as the reflexive, transitive and symmetric closure of the union of the equations generated from each alias set $A1, A2, \ldots$ is ρ. We use the same notation, i.e. $EQ(A)$ to stand for the equations generated from a single alias set. Given an alias set A, we have three possibilities, (i) A consists entirely of identifiers, (ii) A has a single constant or array and (iii) A has several constants or arrays.

In generating $EQ(A)$ we first generate a set of equations from the explicit representation of the alias set and then we close under transitivity, reflexivity and symmetry. The first two cases are easy to handle. Suppose that we have case (i), i.e $A = \{x1, \ldots, xN\}$. Then $EQ(A) = \{x1 = x2, x1 = x3, \ldots, x2 = x3, \ldots\}$. Suppose that we have case (ii) above, with the single non-identifier being c then we proceed as in case (i) except that we add the equations $\{x1 = c, x2 = c, \ldots\}$. In case (iii) we have the possibility of an inconsistency. If we have an inconsistent alias set A, and $\{x1, \ldots, xN\}$ are all the identifiers in A then $EQ(A) = \{x1 = \top, x2 = \top, \ldots, xN = \top\}$. If we have a consistent alias set, then the assumptions of case (iii) require that the terms must all be arrays of the same size or identifiers. For simplicity we consider the case where there are two arrays of size two and no identifiers. If $A = \{[L1, L2], [L3, L4]\}$ then we set $EQ(A) = \{L1 = L3, L2 = L4\}$. If we have identifiers, say x and y in A as well, we add the equations $x = y, x = [L1, L2], y = [L1, L2], x = [L3, L4], y = [L3, L4]$ to $EQ(A)$. If the equations induced by equating array components involve two arrays then the resulting equations are also added to $EQ(A)$. Thus $EQ(A)$ may contain infinitely many equations. It should be clear that $EQ(A)$ is defined to express all the semantic consequences of a given set of equations and is not intended to be an effective procedure. The equations added by unification do not change the meaning of the configurations; they merely change the way the equations are being represented, in other words the relations $\overset{*}{\rightsquigarrow}$preserves the meaning; more precisely, we can prove: if $\rho \overset{*}{\rightsquigarrow} \rho'$ then $EQ(\rho) = EQ(\rho')$. Similarly, the functional environment ρ_F can be converted into a set of equations; denote this set by $EQ(\rho_F)$.

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. In the following the semantic function $\mathcal{M}$ assigns to configurations a closure operator of suitable type. We use the semantic functions $\mathcal{E}, \mathcal{F}$ and $\mathcal{C}$ used in the denotational semantics and the same notational conventions.

$$\mathcal{M}[\![\langle D, e, \rho, \rho_F, FL\rangle]\!] \quad \langle a, env\rangle = \mathbf{lcs}\begin{cases} \langle a, env\rangle \sqsubseteq \langle benv'\rangle \\ env' = \mathcal{C}[\![D \cup \rho \cup \rho_F]\!] \ env' \\ \langle b, env'\rangle = \mathcal{E}[\![e]\!] \ \langle b, env'\rangle \end{cases} \mathbf{in}\ \langle b, env'\rangle$$

The function $\mathcal{M}$ is intended that $\mathcal{M}$ represents the effect of the complete computation on a configuration. The theorem we will prove shows that as we rewrite a configuration the meaning as given by $\mathcal{M}$ will essentially not alter.

In this section, we outline the proof that one step of the reduction relation leaves the first order component of denotation of configurations unaltered. This suffices since we assume that the only observable part of the results of a program are the first order values. Before stating the theorem, we set up notation to ensure that there is no clash of names between the syntactic and semantic environments. This follows the treatment of the first order fragment. Thus, we require that the semantic environment env and the syntactic environment ρ satisfy

$$Dom(env) \cap FL = \emptyset \ldots (II)$$

so that there will be no conflicts occurring when the rewriting needed for array allocation is performed. Also, we use the notation $=_{firstorder}$ to denote equality with respect to the first order components.

The following definitions are useful in the theorem and proofs that follow. Let $a \in G\sigma \rightarrow \tau$, $b \in G_\sigma, c \in G_\tau$. Then,

$$\begin{aligned} DOM(a) &= \{\langle x, y\rangle \mid (\exists)\langle x, y\rangle \rightarrow \langle x', y'\rangle \in a\} \\ DOM(a) \rightarrow DOM(a) &= \{\langle x, y\rangle \rightarrow \langle x, y\rangle \mid \langle x, y\rangle \in DOM(a)\} \\ \langle b, c\rangle \rightarrow \langle b, c\rangle &= \{\langle u, v\rangle \rightarrow \langle u, v\rangle \mid u \sqsubseteq b \wedge v \sqsubseteq c\} \end{aligned}$$

Theorem 3 *Suppose that the following rewrite is possible:*

$$< D, e, \rho_F, \rho, FL > \ \rightarrow \ < D', e', \rho', \rho'_F, FL' >$$

Let env satisfy the following conditions:

- *Condition (II) with respect to both ρ, ρ_F and ρ', ρ'_F.*
- *Forall function symbols F, with definition $F = L$ in one of $e, e', D, D', \rho_F, \rho'F$, $\mathcal{E}[\![F = L]\!]\ env[F \mapsto [DOM(a) \rightarrow DOM(a)]] = env$*

Then, restricted to the first order component of the results, we have

$$\mathcal{M}[\![\langle D, e, \rho, \rho_F, FL\rangle]\!]\langle a, env\rangle = \mathcal{M}[\![\langle D, e, \rho, \rho_F, FL\rangle]\!]\langle a, env\rangle$$

Proof: In order to prove the theorem, we prove the following stronger statement: Let $e^\star$ be any other expression such that $e^\star$ does not define any function symbols defined in the configuration $\langle D, e, \rho, \rho_F, FL\rangle$: alternatively, this can be stated by saying that any function defined in the initial configuration occurs(if at all) only in applicative contexts in $e^\star$. Let

- $< D, e, \rho_F, \rho, FL > \;\rightarrow\; < D', e', \rho', \rho'_F, FL' >$
- $\mathcal{M}[\![\langle D, e, \rho, \rho_F, FL\rangle]\!]\langle a, env\rangle = \langle b, env^\star\rangle$
- $\mathcal{M}[\![\langle D', e', \rho', \rho'_F, FL'\rangle]\!]\langle a, env\rangle = \langle b', env'^\star\rangle$
- *env* is *valid* for $e^\star$.

Then, we have the following:

- $\langle b, env^\star\rangle =_{firstorder} \langle b', env'^\star\rangle$
- $\langle c, env^\star\rangle \in Fix(\mathcal{E}[\![e^\star]\!]) \Leftrightarrow \langle c, env'^\star\rangle \in Fix(\mathcal{E}[\![e^\star]\!])$

The proof proceeds by induction on the size of the proof that the one-step reduction applies. The proofs of all the cases are identical to the case of the first order calculus [9]; the only two exceptions are

1. The rule that replaces the occurrences of names of function symbols by the definitions preserves denotations.
2. The rule that performs "β"-reduction.

These proofs are described in the following two subsections. Hence, the result. ∎.

A.1 "β"-reduction preserves meaning

The rule that is being considered in this subsection is as follows:

$< D, (\lambda x.e_1)e_2, \rho_F, \rho, FL > \;\rightarrow\; < D, y = e_2 \textbf{ in } e_1^*, \rho_F, \rho^*, FL^* >$
where $y \in FL$, $FL^* = FL - \{y\}$
where $e_1^* = e_1[y/x]$, $\rho^* = \rho \cup \{\{y\}\}$

We prove below that $\mathcal{E}[\![(\lambda x.e_1)e_2]\!]$ and $\mathcal{E}[\![x = e_2 \textbf{ in } e_1]\!]$ are "essentially" identical as closure operators.

Lemma 7 *Let* $env[x] = \perp$, *and* x *not free in* e_2. *Then,*

$$\langle a, env\rangle = \mathcal{E}[\![(\lambda x.e_1)e_2]\!]\ \langle a, env\rangle \Leftrightarrow \langle a, env'\rangle = \mathcal{E}[\![x = e_2 \textbf{ in } e_1]\!]\ \langle a, env'\rangle$$

where env' *differs from* env *only at* x.

Proof: Recall that $\mathcal{E}[\![(\lambda x.e_1)e_2]\!]$ was defined as :

$$\mathcal{E}[\![(\lambda x.e_1)e_2]\!]\ \langle a, env\rangle \;=\; \text{lcs}\begin{cases} \langle a, env\rangle \sqsubseteq \langle a', env'\rangle \\ (b, c, a') = App(b, c, a') \\ \langle c, env'\rangle = \mathcal{E}[\![e_2]\!]\ \langle c, env'\rangle \\ \langle b, env'\rangle = \mathcal{E}[\![\lambda x.e_1]\!]\ \langle b, env'\rangle \end{cases} \\ \textbf{in } \langle a', env'\rangle$$

Thus, if $\langle a, env\rangle = \mathcal{E}[\![(\lambda x.e_1)e_2]\!]\ \langle a, env\rangle$, we can deduce that:

$$\begin{aligned} \langle c, env\rangle &= \mathcal{E}[\![e_2]\!]\ \langle c, env\rangle & 1 \\ \langle b, env\rangle &= \mathcal{E}[\![\lambda x.e_1]\!]\ \langle b, env\rangle & 2 \\ (b, c, a) &= App(b, c, a) & 3 \end{aligned}$$

Writing down the equations imposed by $\langle a', env'\rangle = \mathcal{E}[\![x = e_2 \textbf{ in } e_1]\!]\ \langle a', env'\rangle$, we get

$$\begin{aligned} \langle c', env'\rangle &= \mathcal{E}[\![e_2]\!]\ \langle c', env'\rangle & 4 \\ \langle a', env'\rangle &= \mathcal{E}[\![e_1]\!]\ \langle a', env'\rangle & 5 \\ \langle c', env'\rangle &= \mathcal{E}[\![x]\!]\ \langle a', env'\rangle & 6 \end{aligned}$$

We need to prove that the two sets of equations have essentially the same solutions. Equations $1, 4$ are identical.

We first prove that equations $4, 5, 6$ together imply equations $2, 3$. For this, define $\langle b', env''\rangle = \mathcal{E}[\![\lambda x.e_1]\!]\ \langle\langle c', a'\rangle \to \langle c', a'\rangle, env'\rangle$. Since equation 5 holds and $env[x] = c'$, env' is a fixpoint of the functions $UpdateEnv^{e_1}(\langle u, v\rangle \to \langle u, v\rangle)$, for all $\langle u, v\rangle \sqsubseteq \langle c', a'\rangle$. Thus, we can deduce that $env' = env''$. Let $(b'', c'', a'') = App(b', c', a')$. Since $\langle c', a'\rangle \to \langle c', a'\rangle \sqsubseteq b'$, from definition of App, $b' = b''$. We now show that $c' = c'', a' = a''$. From definition of App, this can be deduced from the following statement:

$$[\langle c_1', a_1'\rangle \to \langle c_2', a_2'\rangle \sqsubseteq b' \wedge \langle c_1', a_1'\rangle \sqsubseteq \langle c', a'\rangle] \Rightarrow \langle c_2', a_2'\rangle \sqsubseteq \langle c', a'\rangle$$

The definition of $UpdateGraph^{e_1}(env')(\langle c_1', a_1'\rangle \to \langle c_1', a_1'\rangle)$ is,

$$UpdateGraph^{e_1}(env')(\langle c_1', a_1'\rangle \to \langle c_1', a_1'\rangle) \;=\; \text{lcs}\begin{cases} env'[x \mapsto \overline{c_1'}] \sqsubseteq env^\star \\ \overline{a_1'} \sqsubseteq a_r' \\ \langle a_r', env^\star\rangle = \mathcal{E}[\![e]\!]\ \langle a_r', env^\star\rangle \\ (\langle c_1', a_1'\rangle \hookrightarrow \langle env^\star[x], b\rangle) = b \end{cases} \\ \textbf{in } b$$

Note that $a_1' \sqsubseteq a' = env'[x]$ and $c_1' \sqsubseteq c'$. Thus, in the above $a_r' \sqsubseteq a', env^\star[x] \sqsubseteq c'$. Hence, $\langle c_2', a_2'\rangle \sqsubseteq \langle c', a'\rangle$, and result follows.

We now prove that equations $2, 3$ together imply equation 5. More precisely, we show that if

$$\begin{aligned} \langle b, env\rangle &= \mathcal{E}[\![\lambda x.e_1]\!]\ \langle b, env\rangle & 2 \\ (b, c, a) &= App(b, c, a) & 3 \end{aligned}$$

holds, then $\langle a, env' \rangle = \mathcal{E}[\![e_1]\!]\ \langle a, env' \rangle$, for $env' = env[x \mapsto c]$. From 3, $\langle c, a \rangle \to \langle c, a \rangle \sqsubseteq b$. We deduce from 2 that env' is a fixpoint of $UpdateEnv^{e_1}(\langle u, v \rangle \to \langle u, v \rangle)$, for all $\langle u, v \rangle \in DOM(b)$. In particular, env' is a fixpoint of $UpdateEnv^{e_1}(\langle c_f, a_f \rangle \to \langle c_f, a_f \rangle)$, for all $\langle c_f, a_f \rangle \sqsubseteq \langle c, a \rangle$. From definition of $UpdateEnv^{e_1}$, equation 5 holds. ∎

A.2 Replacing names of function symbols by definitions

The rewrite rule that is studied in this subsection is the following rule, for x a higher order variable. Thus, the rule replaces names of function symbols by definitions from ρ_F.

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

For the rest of this section, we will concentrate on simple contexts $C[]$. The proof for arbitrary contexts is identical and is omitted. Recall that denotations are preserved if and only if the fixpoint sets of the two closure operators are the same. The following lemma is the basic tool to prove that all fixpoints of the denotation of $F = L; y = L(e)$ are fixpoints of the denotation of $F = L; y = F(e)$.

Lemma 8 *Let L be an expression of from $\lambda x.e$. Let env, a_t, a_{arg}, a_r satisfy the following:*

- $\mathcal{E}[\![L = F]\!]env = env$
- $\mathcal{E}[\![L]\!]\ \langle (\langle a_{arg}, a_r \rangle \to \langle a_{arg}, a_r \rangle), env \rangle = \langle a_t, env \rangle$

Let $env_{new} = env[F \mapsto env[F] \sqcup a_t]$. Then,

1. *Forall expressions e' in which F occurs only in an applicative context, i.e all occurrences of F are as subexpressions of $F(e'')$, $\mathcal{E}[\![e']\!]\ \langle b_t, env \rangle = \langle b_t, env \rangle \Rightarrow \mathcal{E}[\![e']\!]\ \langle b_t, env_{new} \rangle = \langle b_t, env_{new} \rangle$*
2. *$\mathcal{E}[\![L = F]\!]\ env_{new} = env_{new}$.*

Proof: The proof involves reasoning about the semantic definitions.

1. We prove that $[\overline{\langle u, v \rangle \to \langle u', v' \rangle} \sqsubseteq a_t \wedge (\langle u, v \rangle \in DOM(env[F])] \Rightarrow \overline{\langle u, v \rangle \to \langle u', v' \rangle} \sqsubseteq env[F]$. From hypothesis of theorem, $\overline{\mathcal{E}[\![L]\!]\ \langle (\langle a_{arg}, a_r \rangle \to \langle a_{arg}, a_r \rangle), env \rangle} = \langle a_t, env \rangle$. From the definition of $\mathcal{E}[\![\lambda x.e]\!]$, $\overline{\langle u, v \rangle \to \langle u', v' \rangle} \sqsubseteq a_t \Rightarrow \overline{v'} \sqsubseteq \Pi_1(\mathcal{E}[\![e]\!]\ \langle v, env[x \mapsto \overline{\{u\}}] \rangle)$. Result follows from definition of $\mathcal{E}[\![F = L]\!]$, since $\langle u, v \rangle \in DOM(env[F])]$.

   The main proof proceeds by induction on the number of occurrences of F. Below, we formalize the induction step, i.e we assume the result for exp and prove it for $F(exp)$. Let $\langle b_t, env \rangle \in Fix(\mathcal{E}[\![exp]\!])$, and $\mathcal{E}[\![F(exp)]\!]\ \langle c_t, env \rangle = \langle c_t, env \rangle$, where $App(env[F], b_t, c_t) = (env[F], b_t, c_t)$. From induction hypothesis, $\langle b_t, env_{new} \rangle \in Fix(\mathcal{E}[\![exp]\!])$. Since $\langle b_t, c_t \rangle \in DOM(env[F])$, using the result above, $App(env[F], b_t, c_t) = App(env_{new}[F], b_t, c_t)$ and result follows.

2. From the hypothesis of the theorem, $\langle env[F], env\rangle \in Fix(\mathcal{E}[\![L]\!])$. Let $\langle a_t, env\rangle \in Fix(\mathcal{E}[\![L]\!])$. Since L is of the from $\lambda x.e$, and L and F are of the same type, all occurrences of F in L occur in an applicative context. From the first part of the theorem, we deduce that $\langle env[F], env_{new}\rangle \in Fix(\mathcal{E}[\![L]\!])$. Similarly, $\langle a_t, env\rangle \in Fix(\mathcal{E}[\![L]\!]) \Rightarrow \langle a_t, env_{new}\rangle \in Fix(\mathcal{E}[\![L]\!])$. Using additivity of L, $\langle env[F]\bigsqcup a_t, env_{new}\rangle \in Fix(\mathcal{E}[\![L]\!])$. Hence, $\mathcal{E}[\![L = F]\!]\ env_{new} = env_{new}$. ∎

Now, we prove a partial converse to the above lemma. That is, we are looking for tools to prove that all fixpoints of the denotation of $F = L; y = F(e)$ are fixpoints of the denotation of $F = L; y = L(e)$. There is a subtlety associated with this proof. The subtleties arise because the semantics does not place sufficient restrictions on the values of the graphs that go into $env[F]$. In particular, a semantic environment env may have an approximation $env[F]$ such that the graph $env[F]$ is not "realized" by the denotation of L. For example, consider the definition $F = \lambda x.x$. Consider the graph $\overline{\{\langle 1, 1\rangle \rightarrow \langle 1, T\rangle\}}$. This graph has information that cannot arise in an operational reduction sequence. Thus, there can be more information in the term $F = L; y = F(e)$ than in the term $F = L; y = L(e)$, making it highly unlikely that we have a converse to the above lemma. So, we identify restrictions on the semantic environment, that allow us to prove a converse. Later, we will show that the semantic environments associated with actual programs do satisfy this restriction.

This is done by defining a notion of a "valid" environment. The motivation behind the definition is as follows. We want to identify environments env (relative to a given expression e) such that the graph $env[F]$ does not contain more information than the definition of the function symbol F in e. The motivation is to exclude pathological cases such as the one alluded to in the above example. Thus, we identify semantic environments env such that the input-output behavior encoded in the graphs of the function symbols match the information contained in their definitions. For example, consider a definition $F = L$ and a semantic environment env. We want every element $\langle u, v\rangle \rightarrow \langle u', v'\rangle$ in $env[F]$ to be attested by L: informally, L on input u with approximation to result v should refine u to u' and v to v'.

Definition 9 *env is* valid *for e, if forall function symbols F such that $F = L$ is a subexpression of e,* $\mathcal{E}[\![F = L]\!]env[F \mapsto [DOM(env[F]) \rightarrow DOM(env[F])]] = env$.

For example, note that the uninitialized environment (the initial environment of the program) $env_\perp$ is valid for any expression. The validity property is "preserved" by the semantics.

Lemma 9 *Let env be* valid *for e and $\mathcal{E}[\![e]\!]\ \langle a, env\rangle = \langle b, env'\rangle$. Then, env' is valid for e.*

Proof: The proof is a simple structural induction on expressions, noting that all elements of form $\langle u, v\rangle \rightarrow \langle u', v'\rangle \in env[F]$ such that $\langle u, v\rangle \neq \langle u', v'\rangle$, "arise" from the subexpression $F = L$ only. ∎

Note that the only environments that we are interested in are those got by executing a program in an uninitialized environment. Thus, lemma 9 allows us to deduce that all the environments that we are interested are *valid* for the relevant expressions.

Lemma 10 *Let L be an expression of form $\lambda x.e$. Let env, a_t satisfy the following conditions:*

- *env is* valid *for $L = F$*
- $\mathcal{E}[\![L]\!]\ \langle DOM(a_t) \to DOM(a_t), env\rangle = \langle a_t, env\rangle$.

Then, for any a_{arg}, b such that $\langle a_{arg}, b\rangle \to \langle a_{arg}, b\rangle)\sqsubseteq a_t$, $App(env[F]\sqcup a_t, a_{arg}, b) = App(a_t, a_{arg}, b)$.

Proof: We prove that $[\overline{\langle u,v\rangle \to \langle u',v'\rangle}\sqsubseteq env[F] \wedge \langle u,v\rangle \in DOM(a_t)]\Rightarrow\overline{\langle u,v\rangle \to \langle u',v'\rangle}\sqsubseteq a_t$. Result of theorem follows immediately from the definition of the function *App*. This is proved by considering the fixpoint iteration that computes $\mathcal{E}[\![F = L]\!]$. Recall that $\mathcal{E}[\![F = L]\!]\ env[F \mapsto [DOM(a_t) \to DOM(a_t)]]$ is defined as

$$\textbf{lcs}\begin{cases} env[F \mapsto [DOM(a_t) \to DOM(a_t)]] \sqsubseteq env' \\ \langle b, env'\rangle = \mathcal{E}[\![L]\!]\ \langle b, env'\rangle \\ env'[F] = b \end{cases}$$
$$\textbf{in}\ env'$$

Unwinding the fixpoint iteration that computes the least common solution, we get

$$\begin{aligned} b^{k+1}, env^{k+1}_{temp} &= \mathcal{E}[\![L]\!]\ \langle b^k, env^k\rangle \\ env^{k+1} &= env^{k+1}_{temp}[F \mapsto [env^{k}_{temp}[F]\sqcup b^{k+1}]] \end{aligned}$$

where $b^0 = DOM(a_t) \to DOM(a_t), env^0 = env[F \mapsto [DOM(a_t) \to DOM(a_t)]]$. In the special case that is being considered here, the fixpoint iteration can be simplified considerably. This is done by proving that $\langle b^{k+1}, env^{k+1}_{temp}\rangle\sqsubseteq\mathcal{E}[\![L]\!]\ \langle DOM(a_t) \to DOM(a_t), env^k\rangle$. The proof proceeds by induction on k. Result is true for $k = 0$. For the induction step, note that

$$\begin{aligned} \langle b^{k+2}, env^{k+2}_{temp}\rangle &= \mathcal{E}[\![L]\!]\ \langle b^{k+1}, env^{k+1}\rangle \\ &\sqsubseteq \mathcal{E}[\![L]\!]\ \langle DOM(a_t) \to DOM(a_t), env^{k+1}\rangle \end{aligned}$$

Thus, the iterates can be rewritten as

$$\begin{aligned} b^{k+1}, env^{k+1}_{temp} &= \mathcal{E}[\![L]\!]\ \langle DOM(a_t) \to DOM(a_t), env^k\rangle \\ env^{k+1} &= env^{k+1}_{temp}[F \mapsto [env^{k+1}_{temp}[F]\sqcup b^{k+1}]] \end{aligned}$$

Given $\overline{\langle u,v\rangle \to \langle u',v'\rangle}$, $(\exists i)\ [\overline{\langle u,v\rangle \to \langle u',v'\rangle}\sqsubseteq env^i[F]]$. From the special form of the fixpoint iteration above, $\langle\langle u,v\rangle \to \langle u',v'\rangle, env^i\rangle\sqsubseteq\mathcal{E}[\![L]\!]\ \langle DOM(a_t) \to DOM(a_t), env^i\rangle$. Note that $env^i\sqsubseteq env$ and $\langle u,v\rangle \in DOM(a_t)$. Thus, using monotonicity of $\mathcal{E}[\![L]\!]$, we can deduce that $\overline{\langle\langle u,v\rangle \to \langle u',v'\rangle, env\rangle}\sqsubseteq\mathcal{E}[\![L]\!]\ \langle DOM(a_t) \to DOM(a_t), env\rangle = \langle a_t, env\rangle$. Hence, the result. ∎

B Adequacy

In this section, we prove that the operational semantics actually attains the values predicted by the denotational semantics. Along with the fact that one-step reduction preserves meaning, this means that the results predicted by the operational and denotational semantics match exactly; this is usually called *adequacy.* Since infinite objects are present in the semantic domain, we cannot say that if the denotational semantics predicts a value, that value is actually attained by a *finite* reduction sequence What we say instead, is roughly speaking, that for every program (first-order term) there is a reduction sequence to every *finite approximant* of the predicted value.

The proof of adequacy uses the idea of logical relations used in the proofs of adequacy for the simply typed lambda calculus [19] to lift the adequacy proof for first order terms to terms of higher types. The rest of this section is organised as follows. In the first subsection, we define inclusive predicates to relate syntactic and semantic entities. In the next subsection, we prove the adequacy theorem.

B.1 Relating syntactic and semantic values

This section is organised as follows. First, we develop tools to relate first order syntactic and semantic values. Next, we extend these tools to accommodate higher type objects.

B.1.1 First order objects

The following definition relates syntactic expressions and semantic values, and syntactic environments and semantic environments. Intuitively, $v \preceq (e, \rho)$ means that that e when evaluated in syntactic environment ρ gives a value that is more defined than v. $env \preceq \rho$ can be viewed as saying that the syntactic environment ρ is more constrained than env. The third case of the definition combines the first two cases in a natural way.

Definition 10 *Syntactic values and environments are related to semantic values and environments as follows:*

1. e covers v in ρ *, written* $v \preceq (e, \rho)$*, if* ρ *consistent implies that one of the following holds:*
   (a) v *is a basic value, and* $\langle \emptyset, e, \rho_F, \rho, FL\rangle \xrightarrow{*} \langle \emptyset, v, \rho_F, \rho, FL\rangle$.
   (b) $v = a$*, where* a *is of type array, and* $(a\langle s\rangle = v') \Rightarrow (\langle \emptyset, e\langle s\rangle, \rho_F, \rho, FL\rangle \xrightarrow{*} \langle \emptyset, v', \rho_F, \rho, FL\rangle)$*, where* v' *is a basic value, and* s *is any finite sequence.*
2. ρ covers env*, written* $env \preceq \rho$*, if for all variable names* x*,* $env[x] \preceq (x, \rho)$
3. $\langle e, \rho\rangle$ covers r*, written* $r \preceq \langle e, \rho\rangle$*, if* $env \preceq \rho$ *and* $v \preceq (e, \rho)$.

Note that an inconsistent environment ρ is defined to dominate all semantic environments env. Furthermore, if $env = env_\top$, and $env \preceq \rho$, then ρ is inconsistent.

For notational convenience, we follow the convention that the syntactic environment associated with the operational configuration *error* is inconsistent. Furthermore, we denote finite elements of the semantic domains by the subscript $_f$. For example, a finite element of the value domain will usually be denoted by a_f or b_f. A finite element of ENV will usually be denoted by env_f and a finite element of $V \times ENV$ will usually be denoted by r_f. First, we define a relationship between expressions e and closure operators f on $V \times ENV$. Roughly speaking, $f \preceq e$ means that when e is evaluated in a suitable syntactic configuration, the resulting expression has a meaning that dominates the result predicted by f. In the following definition, we use # in the expression part of the configuration to indicate that the actual expression is not relevant to the definition. The use of $\star$, $\star\star$ and so on is motivated by the same reason.

Definition 11 *$f \preceq_x e$ is defined as follows. Let*

- $f\langle a_f, env\rangle = r$
- $env \preceq \rho$
- $a_f \preceq (x, \rho)$

Then, given $\langle\{x = e\}, \#, \star, \star\star, \star\star\star\rangle \overset{*}{\longmapsto}_s \langle D, \#, \rho_F, \rho, FL\rangle$, *where x is any variable name,*

$$(\forall r_f \sqsubseteq r)\ (\exists)\ [\langle D, \#, \rho_F, \rho, FL\rangle \overset{*}{\longrightarrow} \langle D', \#, \rho'_F, \rho_{res}, FL'\rangle \wedge r_f \preceq \langle x, \rho_{res}\rangle]$$

The special case of the above definition that we are interested in is when $a_f = \perp$, and $\langle\{x = e\}, x, \star, \star\star, \star\star\star\rangle = \langle D, x, \rho_F, \rho, FL\rangle$, and $f = \mathcal{E}[\![e]\!]$. As before, the greater generality of the definition simplifies the proofs.

Now, we have all the tools to define a relationship between expressions e and closure operators f on $V \times ENV$. Roughly speaking, $f \preceq e$ means that when e is evaluated in a suitable syntactic configuration, the resulting expression has a meaning that dominates the result predicted by f.

Definition 12 $f \preceq e \Leftrightarrow (\forall x)\ [f \preceq_x e]$ *as follows.*

Let ρ be more constrained than env. Let $\mathcal{E}[\![e]\!]\ env\ \perp = \langle env', b\rangle$. Then, given any finite approximant $\langle b_f, env_f\rangle$ to $\langle b, env'\rangle$, there is a finite reduction sequence evaluating expression e in syntactic environment ρ such that the resulting syntactic environment ρ_{res} is more constrained than env_f, and the resulting expression e' evaluated in ρ_{res} yields a more defined value than b_f. In particular, if env' is the error environment, evaluating e in ρ results in *error*.

B.1.2 Higherorder objects

In this section, we extend the relation $\preceq$ to work for higher order objects. This is done following the idea of logical relations used in the adequacy proofs for typed functional languages [19]. The idea is to "lift" the first order definition to expressions of higher type. The following definition proceeds by induction on formation of types. The base case of the definition uses the inclusive predicate defined in the previous subsection. Define a predicate $Comp$ by induction on types, on closed terms as follows:

Definition 13 *(Definition of the computability predicate)*

- *$Comp(e)$ is true, for an expression e of base type if $\mathcal{E}[\![e]\!] \preceq e$*
- *$Comp(e)$ is true, for an expression e of type $\sigma_1 \rightarrow \sigma_2$, if $(\forall e')$, e' of type σ_1, such that $Comp(e')$, $Comp(e(e'))$*

The definition is extended to open terms through the notion of valid substitution. Valid substitutions capture the right syntactic conditions for binding variables in a context of constraints.

Definition 14 *(Valid Substitutions)*
Let $\{x_1^{\sigma_1} \ldots x_n^{\sigma_n}\}$ be a set of variables, Then, a set of equations E is a valid substitution for $\{x_1^{\sigma_1} \ldots x_n^{\sigma_n}\}$ if

- *E has no free variables.*
- *All non-first order variables x_i occur as the left hand side exactly one equation.*
- *All first order variables x_i occurs as the left hand side of at least one equation.*
- *Every equation $y_j = e_j$, where y_j is of non-base type satisfies $Comp(e_j) \vee [Comp(y_j) \Rightarrow Comp(e_j)]$.*

Now, we can define the computability predicate for terms with free variables.

Definition 15 *Let e be an open term with $FV(e) = \{x_1 \ldots x_n\}$. Then, $Comp(e)$ if forall valid substitutions E, $Comp(E$* **in** *$e)$.*

B.2 Adequacy Proof

In this section, we prove that the main theorem: the operational semantics actually attains the values predicted by the denotational semantics. We show that any *finite approximant* of the predicted value can be produced by a finite reduction sequence. The proof proceeds by structural induction on the formation of terms using the computability predicate defined earlier.

The cases of structural induction for the first order combinators has been proved in earlier work [9]. This lemma is the base case for the adequacy proof for the higher order language. The intuition behind the proof is as follows: Consider the case of composition of definitions. The aim to construct a reduction sequence from semantic information. In this case, we use the special properties of fixed points of closure operators to carry out this construction. In some sense, this is the key to the whole adequacy proof. Suppose that g_1 and g_2 are two closure operators on ENV that correspond to the imposition of two constraints given as sets off 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.

Lemma 11 *(First order properties)*

- *Variables of base type are computable.*
- $Comp(e) \Rightarrow Comp(array(e))$
- $Comp(e_1) \wedge Comp(e_2) \Rightarrow Comp(e_1[e_2])$
- $Comp(e_1) \wedge Comp(e_2) \Rightarrow Comp(e_1 \; op \; e_2)$
- $Comp(e_1) \wedge Comp(e_2) \wedge Comp(e_3) \Rightarrow Comp(cond(e_1, e_2, e_3)$

As the next step of the proof, we prove that variables of higher type are computable. The proof is non-trivial, because of the implicit recursion in equations defining variables of higher type: recall that the syntax permitted definitions of the type $F = L[F]$. The lemma below is the key piece in the proof: it ¡proves (the unsurprising fact) that any finite piece of the result got by evaluating such a recursive definition is got by unwinding the definition finitely many times. In the statement and proof of the following lemma $L[F]$ is used as notation for a term with possible free occurrences of F.

Lemma 12 *Let*

- $\mathcal{E}[\![F = L[F]]\!] \; env[F \mapsto [DOM(env[F]) \rightarrow DOM(env[F])] = env$
- $\overline{\langle u, v\rangle \rightarrow \langle u', v'\rangle} \sqsubseteq env[F]$

Then, there is an n such that $\overline{\langle u, v\rangle \rightarrow \langle u', v'\rangle} \sqsubseteq env'[F_n]$, *where*

$\mathcal{E}[\![F_0 = \lambda x.x; F_1 = L[F_0]; \ldots F_n = L[F_{n-1} \ldots]\!] \; env[F_i \mapsto [DOM(env[F]) \rightarrow DOM(env[F])]] = env'$

Proof: The proof proceeds by unwinding the fixpoint iteration that computes the result got by applying $\mathcal{E}[\![F = L[F]]\!]$ to $env[F \mapsto [DOM(env[F]) \rightarrow DOM(env[F])]$. A stage in the fixed point iteration is as follows:

$$\begin{aligned} \langle b^{k+1}, env^{k+1}_{temp}\rangle &= \mathcal{E}[\![L[F]]\!] \; \langle b^k, env^k\rangle \\ env^{k+1} &= env^{k+1}_{temp}[F \mapsto b^{k+1} \bigsqcup env^k[F]] \end{aligned}$$

where $b^0 = DOM(env[F]) \rightarrow DOM(env[F]), env^0 = env[F \mapsto [DOM(env[F]) \rightarrow DOM(env[F])]]$.

In this special case, the fixed point iteration can be simplified. This is done by proving on induction on k that $b^{k+1}, env_{temp}^{k+1} \sqsubseteq \mathcal{E}[\![L[F]]\!] \langle DOM(env[F]) \rightarrow DOM(env[F]), env^k \rangle$. This proof is straightforward and is omitted. Thus, the equations can be simplified to

$$\begin{aligned} \langle b^{k+1}, env_{temp}^{k+1} \rangle &= \mathcal{E}[\![L[F]]\!] \langle DOM(env[F]) \rightarrow DOM(env[F]), env^k \rangle \\ env^{k+1} &= env_{temp}^{k+1}[F \mapsto env_{temp}^{k+1}[F] \sqcup b^{k+1}] \end{aligned}$$

We prove that $env^k[F] \sqsubseteq env'[F_k]$. The proof proceeds by induction on k. The statement of the inductive step is: $env^k[F] \sqsubseteq env'[F_k] \Rightarrow$

- $\langle b^{k+1}, env^{k+1}[F \mapsto \perp] \rangle \sqsubseteq \mathcal{E}[\![L[F_k]]\!] \langle DOM(env[F]) \rightarrow DOM(env[F]), env' \rangle$
- $env^{k+1}[F] \sqsubseteq env'[F_{k+1}]$

Note that $env^0[F] = env'[F_0] = DOM(env[F]) \rightarrow DOM(env[F])$. Thus, the base case is true. We prove the inductive step here. From the simplified form of the fixpoint iteration proved above: $\langle b^{k+1}, env^{k+1}[F \mapsto \perp] \rangle \sqsubseteq \mathcal{E}[\![L[F]]\!] \langle DOM(env[F]) \rightarrow DOM(env[F]), env_k \rangle$. All occurences of F in $L[F]$ are in contexts of the form $F(e')$. Thus, from hypothesis $env^k[F] \sqsubseteq env'[F_k]$, we get $\langle b^{k+1}, env_{temp}^{k+1}[F \mapsto \perp] \rangle \sqsubseteq \mathcal{E}[\![L[F_k]]\!] \langle DOM(env[F]) \rightarrow DOM(env[F]), env' \rangle$.

Given $\langle u, v \rangle \rightarrow \langle u', v' \rangle \sqsubseteq env[F]$, from the continuity of all functions involved, there is an n such that $\langle u, v \rangle \rightarrow \langle u', v' \rangle \sqsubseteq env^n[F] \sqsubseteq env'[F_n]$. ∎

Lemma 13 *Variables of non-base type are computable.*

Proof: The only non-trivial case to consider is to prove that expressions of the form $F = L[F]$ **in** F are computable. Consider the expressions $F_0 = \lambda x.x; F_1 = L[F_0]; \ldots F_n = L[F_{n-1}]$, for every n. Without loss of generality, assume that there are no bindings for F_i's in $D \cup \rho_F$ below. Then,

1. From definition of valid substitutions, it follows that, forall n, the terms $F_0 = \lambda x.x; F_1 = L[F_0]; \ldots F_n = L[F_{n-1}]$ **in** F_n are computable
2. $\langle D[F_n], e[F_n], \rho, \rho_F^{\star}, FL \rangle \xrightarrow{*} \langle D'[F_0], e[F_0], \rho, \rho_F^{\star\star}, FL \rangle$ implies that
$$\langle D[F], e[F], \rho, \rho_F, FL \rangle \xrightarrow{*} \langle D'[F], e[F], \rho, \rho'_F, FL \rangle$$
where the starred ρ_F's differ from the the corresponding unstarred ρ_F's only in the bindings of $F, F_0 \ldots F_n$: the proof is a straightforward inductive argument on the length of the reduction sequences.

Consider a sequence of expressions $e_1 \ldots e_n$ such that $(F = L[F]$ **in** $F)\ e_1 \ldots e_n$ is of ground type. Let env be *valid* for $(F = L[F]$ **in** $F)\ e_1 \ldots e_n$, in the notation of the previous section. Let $\mathcal{E}[\![(F = L[F]$ **in** $F)\ e_1 \ldots e_n]\!]\langle a, env \rangle = \langle b, env' \rangle$. Given any finite approximant r to the result, there is an m such that

$$r \sqsubseteq \mathcal{E}[\![(F_0 = \lambda x.x; F_1 = L[F_0]; \ldots F_m = L[F_{m-1}] \textbf{ in } F_m)\ e_1 \ldots e_n]\!]\langle a, env \rangle$$

Result follows from the two observations above. ∎

Theorem 4 *All terms are computable.*

Proof: Proof is by structural induction. From the lemmas above, all variables are computable. From lemma 11 all first order constructors form computable expressions from computable expressions. The cases of structural induction for E **in** e and $\lambda x.e$ and $e(e')$ follow straightforwardly from definitions. ∎