A Fully Abstract Semantics for a Functional Language with Logic Variables

Radha Jagadeesan
Prakash Panangaden
Keshav Pingali*

TR 89-969
May 1989

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

*This research is supported by NSF Grant CCR-8702668, and by grants from the Math Sciences Institute, Cornell and from the General Electric Corporation.

A Fully Abstract Semantics for a Functional Language with Logic Variables

Radha Jagadeesan Prakash Panangaden Keshav Pingali *
Computer Science Department
Cornell University

May 9, 1989

Abstract

We present a novel denotational semantics for a functional language with logic variables intended for parallel execution. The intuition behind this semantics is that equations represent equational constraints on data. Thus a system of equations can be viewed as defining a set of, possibly inconsistent, constraints. The semantics is couched in terms of closure operators on a Scott domain. This allows one to abstract away from all the complexities associated with operational reasonong expressed in terms of concurrent threads of execution. We define a structural operational semantics for the language that expresses precisely the concurrent execution model that we have in mind. We show that the abstract denotational semantics is fully abstract with respect to the operational semantics. This is surprising given how very different the two semantic descriptions are. It also shows that thinking in terms of constraints is an accurate substitute for thinking in terms of explicit parallel execution. The proof of full abstraction is complicated by the fact that there are potentially infinite objects in the domain.

*Keshav Pingali is supported by NSF grant CCR-8702668, and by grants from the Math Sciences Institute, Cornell and from the General Electric Corporation.

1 Introduction

Programming languages can be divided into two categories: imperative and declarative. The semantics of imperative languages like Fortran and Pascal rely on an updatable global store and are tied intimately to the von Neuman model of sequential computation. In contrast, declarative languages, such as functional and logic programming languages, can be given semantics using abstractions such as values, functions and relations. This has two advantages. Clean semantics result in simple proof rules; hence, declarative language programs are easier to verify than imperative language programs. More importantly, the absence of sequentiality in the semantics of declarative languages makes these languages promising candidates for programming parallel machines. For these reasons, much research has been done in both functional and logic programming languages over the past decade.

One area of special interest is the integration of functional and logic programming. From the viewpoint of logic programming, such an integration is mandatory for efficiency. Although anything computable can be computed using pure Horn clauses, the represention of integers, for instance, as terms built from the functors zero and successor is not recommended for programmers who worry about how fast their programs run. Every 'real' logic programming language includes functions for doing arithmetic and logical operations, as well as an operator (such as *is* or :=) for binding an identifier to the result of performing such an operation. We consider these to be functional constructs that have been grafted on to a logic programming language. From the viewpoint of functional programming, the utility of such an integration is somewhat more subtle. Logic programming offers two features that are not present in functional languages: automatic back-tracking (or OR-parallelism, its analog in parallel logic programming languages) and the logic variable (*i.e.*, variables that are bound incrementally by constraint intersection). Unfortunately, it is difficult to implement OR-parallelism efficiently and the logic programming community is investigating a number of alternatives such as commited choice non-determinism. At this point, there is no consensus about the best alternative and it does not seem very fruitful to incorporate any form of OR-parallelism into functional languages.

Logic variables, on the other hand, can be introduced quite easily into a functional language (basically, by replacing pattern-matching by unification) and the merits of doing so have been remarked on by many researchers [2]. Among other things, logic variables permit elegant coding of constraint-based algorithms such as Milner's polymorphic type deduction algorithm and of symbol-table management algorithms in compilers. Our interest in integrating logic variables into a functional language arises from the observation that logic variables can be used to define data structures incrementally. In a pure functional language, a data structure is a value (just like an integer or floating point number) which is produced as the result of evaluating a single applicative expression. This is satisfactory when the data structure is built bottom-up (as lists are): first, the components of the data structure can be constructed, and then these components can be assembled together to produce the desired data structure. However, this does not work for 'flat' data structures, such arrays and matrices which are very important in many problem domains such as scientific computation where they are used in finite-element calculations to hold the values of physical variables such as pressure or temperature. Constructing such arrays and matrices functionally is difficult because usually, there is no uniform rule for computing matrix elements; for example, the computation of boundary elements may be quite different from the computation of interior elements. In such situations, writing a single applicative expression for defining the entire matrix can be inefficient and the resulting program may be quite obscure. An alternative is to compute the desired matrix as the limit of a sequence of matrices which differ incrementally from each other. Unfortunately, the absence of an update operation in functional languages means that each matrix in this sequence is a different value, and the construction of a matrix of size $n \times n$ may involve making n^2 copies of the matrix! Logic variables provide an elegant solution to this problem because they allow the programmer to define an array incrementally without making numerous intermediate copies. To construct a large matrix, the programmer first allocates a matrix of the desired size. Each element of this matrix can be thought of as containing an uninitialized logic variable. These logic variables can be bound incrementally in the program; for example, the array can be passed to two procedures, one of which instantiates variables on the boundary while the other instatiates variables in the interior. In this way, large data structures can be constructed without the copy overhead of functional data structures. This is similar to the use of difference-lists in pure logic programming.

These observations motivated the design of *Id Nouveau*, which is a functional language with arrays that behave like first-order terms in logic programming languages[6]. 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. In this paper, we provide a formal description of the first-order subset of Id Nouveau. This subset is introduced informally in Section 2 through two programming examples which illustrate the subtle interaction between logic variables and concurrency. To concentrate on the essentials, we define in Section 3 a core language called Cid. Any first-order Id Nouveau program can be translated into a Cid program in a straight-forward manner. In Section 4, we give a Plotkin-style structural operational semantics for Cid. The operational semantics is an interleaving of reduction (as in functional languages) and constraint-solving (through unification). The interaction between concurrency and logic variable instantiations (which are like globally visible side-effects) is fairly subtle and it is a non-trivial problem to give an abstract denotational semantics for this language. In Section 5, we present a denotational semantics for Cid which abstracts away from operational details and which is couched in terms of equation solving. In Section 6, we show that the denotational semantics is fully abstract with respect to the operational semnatics [4,7] thus showing that the abstract semantics fits precisely with the concrete operational semantics. This last section is rather long and contains the hard technical proofs that are needed to establish full abstraction.

2 Informal Introduction to the Language

Id Nouveau can be thought of as a functional language augmented with array manipulation primitives from logic programming languages. This section introduces the language and the operational semantics informally through a number of programming examples. The operational semantics we present in this section is a simplified version of the formal operational semantics in Section 4. For a complete description of Id Nouveau, we refer the reader to [6]. We assume that the reader is familiar with functional languages; therefore, we will begin by describing the constructs for manipulating arrays. The programs in this section illustrate the differences between functional and logical arrays, and also highlight the subtle interaction between concurrency and logic variable instantiation.

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]
```

To put these constructs together and to introduce our operational model, we discuss a program to solve the inverse permutation problem: given an array `B` of lengthn containing a permutation of the integers 1..n, build a new array `A` of lengthn 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}
```

As in functional languages, 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. In the first step, the body of the function is expanded out, and the actual parameters are substituted for the formals. This results in the 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)` $\rightarrow$ `[L1,...,Ln]` where the identifiers `L1,...,Ln` are new identifiers. Intuitively, this rule models the allocation of an array of lengthn in which each element is a distinct, uninstantiated 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]` $\rightarrow$ `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 length3. 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. 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!).

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 some 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,5);
     in A}

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

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

This program produces an array of length10 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.

3 Cid: a subset of Id Nouveau

Id Nouveau is a fairly large language since it is intended to be a real programming language. For the purpose of this paper, we define a core of this language, called Cid, which is rich enough that any Id Nouveau program can be translated straight-forwardly into a Cid program. Working with Cid reduces the number of cases to be considered for the operational and denotational semantics.

```
program ::=
            def F1(id) = {def-list in exp}
            def F2(id) = {def-list in exp}
            ....
            def Fn(id) = {def-list in exp}
            exp

def-list ::= def | def;def-list
```

```
def ::= id = exp

expression ::= constant | id | exp1 op exp2 |
               cond(exp1,exp2,exp3) |
               array(exp) | exp1[exp2] | Fi(exp1)
```

Figure 1: Syntax of Cid

Figure 1 describes the syntax of Cid. The main differences between Id Nouveau and this core language are as follows. The loop construct is eliminated since a loop can be replaced by a tail recursive procedure. In Id Nouveau, some procedures, such as `inverse-permute` return a result, while others, such as `fill-even` are 'pure side-effect' procedures that instantiate variables in their arguments but do not return any results. To simplify notation, we will require that all procedures return a result (a procedure 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 identifer. With these modifications, the `fill-even` procedure becomes

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

To side-step issues regarding the scopes of variables, we follow the logic programming convention: the body of a procedure is a single scope and the formal parameters of the procedure are in the same scope. Finally, to eliminate the need for ellipses and subscripts, we will require that a procedure have exactly one formal parameter and that its body have exactly one local variable. For example, the three parameters of procedure `new-fill-even` can be eliminated in favor of a single parameter `A`, and occurrences of `X,h` and `i` in the body of the procedure are replaced by `A[1]`, `A[2]` and `A[3]` respectively. Thus, the three parameters get replaced by an array of three elements. In the same way, a procedure body can be transformed so that there is exactly one local variable. If F is a function, arg_F, local_F, defs_F and exp_F denote the formal parameter, local variable, definitions in the body and the return expression of F.

4 Operational Semantics of Cid

In this section, we give an operational semantics for Cid using Plotkin-style rewrite rules. First, we give an informal introduction to the operational semantics. Rather than rewrite expressions directly (as we did in Section 2), it is convenient to work with *configurations.* A configuration is a triple $< D, exp, \rho, FL >$ where D is a set of definitions, exp is an expression, ρ is the syntactic environment and FL is the free-list. Intuitively, D contains definitions whose right-hand sides have not yet been completely 'reduced' to a *base value* - that is, an identifier, constant or array. The syntactic environment ρ is a non-empty set of *alias-sets* where an alias-set is an equivalence class of base values. For example, $\{x, y, z\}, \{x, y, 4\}$ 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.

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 to a base value, the definition is removed from D and unified with the environment. If unification fails, the configuration is rewritten to 'Error' and computation aborts. Otherwise, the resulting environment replaces the old one in the configuration, and rewriting continues.

4.1 Syntactic Categories

We define some syntactic categories required for the operational semantics.

$x, L \epsilon$ Id = countable set of identifiers

$c \epsilon$ Constant = set of constants

$Ar\epsilon$ Array $::= [x_1, ..., x_n]$

$B\epsilon$ Base-value $::= x|c|Ar$

$A\epsilon$ Alias-set $::= \{B_1, ..., B_n\}$

$\rho\epsilon$ Environment $::= \phi|\{E_1, ..., E_n\}$

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

$e\epsilon$ expression (defined by the syntax of the language)

$D\epsilon$ Defs $::= \phi|\text{def}_1, \ldots, \text{def}_n$

$C\epsilon$ Configurations $::= < D, e, \rho, FL > |Error$

The notation $[x_1, ..., x_n]$ for arrays represents a sequence of one or identifiers. The *length* of an array is the number of elements in this sequence.

4.2 Unification

The unification algorithm we use is similar to the one in Qute[8]. 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;
```

In this program, `x` is intended to be an 'infinitely nested' array. The unification algorithm we discuss in this section respects this intended meaning.

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 $\leadsto$ on environments.

Definition 2. $\leadsto$ is a binary relation on environments defined as follows:

1. If A1 and A2 are members of an enviroment ρ, and A1 and A2 have an identifier in common, then $\rho \leadsto (\rho$ - $\{A1\}$ - $\{A2\}) \cup \{A1 \cup A2\}$.

2. If $\{[x_1, ..., x_n], [y_1, ..., y_n]\} \subseteq A\epsilon\rho$ then $\rho \leadsto \rho \cup \{\{x_1, y_1\}, ..., \{x_n, y_n\}\}$.

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 $\rho_1 \leadsto \rho_2$ 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 $\overset{*}{\leadsto}$be the reflexive and transitive closure of $\leadsto$.

Theorem 1. The relation $\overset{*}{\leadsto}$has the following properties [8]:

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

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

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

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 quadruple $< D, e, \rho, FL >$. 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 a syntactic environment and A is an alias-set, let $\mathcal{U}(\rho, A)$ denote the unique, irreducible environment such that $(\rho \cup \{A\}) \overset{*}{\rightsquigarrow} \rho_1$. we will say that $\mathcal{U}(\rho, A)$ is the result of unifying ρ and A.

4.3 Rewrite rules

The rewrite rules for configurations are specified in terms of a binary relation $\longrightarrow$ 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 $< \phi, exp_P, \phi, Id >$. In this configuration, D, the set of definitions to be reduced, is empty. In the initial environment, the environment is the empty set 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. Can we view the syntactic environment ρ the same way? 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 of ρ is an element of exactly one alias-set. This leads to the following definition.

Definition 4. If $< D, e, \rho, FL >$ is a configuration and x is an identifier not a member of FL, let A be the (unique) alias-set that contains x. The function ρ(x) is defined by cases on A:

1. All the elements of A are identifiers. In that case, ρ(x) is undefined.

2. 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 ρ(x) to be c.

3. At least one element of A is an array. Since A is consistent, the elements of A are either identifers or arrays of the same length. $\rho(\mathrm{x})$ could be defined to be any one of these arrays. To be precise, place a lexicographical ordering on identifiers and let $\rho(\mathrm{x})$ be the array whose first element is the least in this ordering.

The intuition behind this definition is the following. During the rewrite process, occurrences of an identifier x will be replaced by $\rho(x)$ if $\rho(x)$ is defined. 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 $\rho(x)$ 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 $\rho(x)$ unique by our (fairly arbitrary) condition.

The Plotkin-style operational semantics for Cid is given in Figure 2. Most of the 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.

With this explanation, the rule for function application should be clear. A function application $F(e)$ is rewritten by replacing it with $\exp_F$, and adding the definitions in defs_F to the definitions in D. The formal parameter and local variable of the function are first renamed to new identifiers (say `x` and `y` respectively) to avoid name clashes. Since the actual parameter `e` need not be a base value, a definition `x = e` is added to the definitions in the configuration.

4.4 Properties of the Rewrite Rules

Theorem 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 $C_i \rightarrow_s C_{i+1}$, $W(C_{i+1}) < W(C_i)$. This establishes the required result. Informally, the weight of a configuration $< D, e, \rho, FL >$ 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. More precisely,

$$W(< D, e, \rho, FL >) = C(D) + C(e)$$

$$C(D_1, ..., D_n) = C(D_1) + ... + C(D_n)$$
$$C(id = exp) = C(id) + C(exp)$$
$$C(constant) = 0$$
$$C(id) = 1$$
$$C(e1\ op\ e2) = 2 + C(e1) + C(e2)$$
$$C(cond(e1, e2, e3)) = 2 + C(e1) + C(e2) + C(e3)$$
$$C(array(e)) = 2 + C(e)$$
$$C(e1[e2]) = 2 + C(e1) + C(e2)$$
$$C(F(exp)) = C(exp)$$

It is straight-forward to verify that if $C1 \rightarrow_s C2$, then $W(C2) < W(C1)$.

Expressions:
Identifiers:
1. $< D, x, \rho, FL > \;\rightarrow\; < D, \rho(x), \rho, FL >$ (if $\rho(x)$ is defined)

Basic Operations:

1. $$\frac{< D, e_1, \rho, FL > \;\rightarrow\; < D^*, e_1^*, \rho^*, FL^* >}{< D, e_1 \; op \; e_2, \rho, FL > \;\rightarrow\; < D^*, e_1^* \; op \; e_2, \rho^*, FL^* >}$$

2. $$\frac{< D, e_2, \rho, FL > \;\rightarrow\; < D^*, e_2^*, \rho^*, FL^* >}{< D, e_1 \; op \; e_2, \rho, FL > \;\rightarrow\; < D^*, e_1 \; op \; e_2^*, \rho^*, FL^* >}$$

3. $< D, m \; op \; n, \rho, FL > \;\rightarrow\; < D, r, \rho, FL >$ (where r = m op n)

Conditional:

1. $$\frac{< D, e_1, \rho, FL > \;\rightarrow\; < D^*, e_1^*, \rho^*, FL^* >}{< D, cond(e_1, e_2, e_3), \rho, FL > \;\rightarrow\; < D^*, cond(e_1^*, e_2, e_3), \rho^*, FL^* >}$$

2. $< D, cond(true, e_2, e_3), \rho, FL > \;\rightarrow\; < D, e_2, \rho, FL >$

3. $< D, cond(false, e_2, e_3), \rho, FL > \;\rightarrow\; < D, e_3, \rho, FL >$

Array:

1. $$\frac{< D, e, \rho, FL > \;\rightarrow\; < D^*, e^*, \rho^*, FL^* >}{< D, array(e), \rho, FL > \;\rightarrow\; < D^*, array(e^*), \rho^*, FL^* >}$$

2. $< D, array(n), \rho, FL > \;\rightarrow\; < D, [L1, ..., Ln], \rho^*, FL^* >$
(where L1,...,Ln ϵ FL
$\rho^* = \rho \cup$ {{L1},...{Ln}}
FL^* = FL - {L1,...Ln})

Array Selection:

1. $$\frac{< D, e_1, \rho, FL > \;\rightarrow\; < D^*, e_1^*, \rho^*, FL^* >}{< D, e_1[e_2], \rho, FL > \;\rightarrow\; < D^*, e_1^*[e_2], \rho^*, FL^* >}$$

2. $$\frac{< D, e_2, \rho, FL > \;\rightarrow\; < D^*, e_2^*, \rho^*, FL^* >}{< D, e_1[e_2], \rho, FL > \;\rightarrow\; < D^*, e_1[e_2^*], \rho^*, FL^* >}$$

3. $< D, [L1, ..., Ln][i], \rho, FL > \;\rightarrow\; < D, Li, \rho, FL >$ (where $1 \leq$ i $\leq$ n)

Application:

1. $< D, F(e), \rho, FL > \;\rightarrow\; < D^*, e_1, \rho^*, FL^* >$
(where $D^* = D \cup \{x = e\} \cup defs_F[x/arg_F][y/local_F]$ $\qquad (x, y \epsilon FL)$
$e_1 = exp_F[x/arg][y/local_F]$
$\rho^* = \rho \cup \{\{x\}, \{y\}\}$
$FL^* = FL - \{x, y\})$

Definitions:

1. $$\frac{< D, e, \rho, FL > \rightarrow < D^*, e^*, \rho^*, FL^* >}{< D \cup \{x = e\}, e_1, \rho, FL > \rightarrow < D^* \cup \{x = e^*\}, e_1, \rho^*, FL^* >}$$

2. $< D \cup \{x = y\}, e, \rho, FL > \rightarrow < D, e, \mathcal{U}(\rho, \{x, y\}), FL >$
(if $\mathcal{U}(\rho, \{x, y\})$ is consistent)

$< D \cup \{x = y\}, e, \rho, FL > \rightarrow Error$ (otherwise)

3. $< D \cup \{x = c\}, e, \rho, FL > \rightarrow < D, e, \mathcal{U}(\rho, \{x, c\}), FL >$
(if $\mathcal{U}(\rho, \{x, c\})$ is consistent)

$< D \cup \{x = c\}, e, \rho, FL > \rightarrow Error$ (otherwise)

4. $< D \cup \{x = [L1, ..., Ln]\}, e, \rho, FL > \rightarrow < D, e, \mathcal{U}(\rho, \{x, [L1, ..., Ln]\}), FL >$
(if $\mathcal{U}(\rho, \{x, [L1, ..., Ln]\})$ is consistent)

$< D \cup \{x = [L1, ..., Ln]\}, e, \rho, FL > \rightarrow Error$ (otherwise)

5 Denotational Semantics

As we have discussed in the previous sections, the way to think about programs that use logic variables is in terms of constraints. Thus a definition of the form $x = e$ is viewed as providing a constraint on x. Given this view, the meaning of a constraint is the set of values satisfying the constraint. Two questions then naturally arise; what are appropriate sets of values, and what is the ordering between such sets?

Normally one thinks of defining a powerdomain to describe sets of values taken from a domain. Indeed, the Smythe powerdomain [10], 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, $\langle \bot, \bot \rangle$ satisfies the constraint but $\langle 2, \bot \rangle$ does not satisfy the constraint.

The basic mechanism by which constraints get imposed in Cid is through unification. Each time unification is performed new constraints are imposed on some variables. This always add information, thus we need to describe the imposition of a constraint via a function that adds to the "information content" of its argument. Such functions are just *extensive* functions, i.e. functions that satisfy $\forall x.x \sqsubseteq f(x)$. Clearly we want these functions to be monotonic and continuous as well since the process of generating constraints is supposed to be computable. A final natural requirement is that the imposition of the same constraint a second time has no further effect. Thus the functions modeling the imposition of constraints should be idempotent. Such functions are called closure operators.

The set of fixed points of a closure operator, f, on a domain D, i.e. the set $\{f(x) | x \in D\}$, is the set of values that satisfy the constraint expressed by f. It turns out that ranges of closure operators are the simplest way to characterize the sets that arise. These are discussed in Scott's paper "Data Types as Lattices" [9].

Definition 5. A *closure operator*, f, on a domain V is a continuous function satisfying, (i) $\forall x \in V.x \sqsubseteq f(x)$, (ii) $f \circ f = f$.

The least closure operator on a domain is the identity function. The collection of closure operators themselves forms a complete partial operator. An important point about closure operators is that one can find common fixed points for any number of closure operators at once. The construction is quite simple. Suppose that f and g are closure operators on some domain D. The function $f \circ g$ is also extensive, continuous and monotonic. The least fixed point of $f \circ g$ is the least common fixed point of f and g. It is easy to check this by a simple calculation.

The domain of values that we use in modeling Cid is the domain of nested arrays. We give a formal account of this domain in the next subsection. Intuitively, the domain, called V, has a top element T to model inconsistent constraints. The arrays have arbitrary finite length and each array may have components that are fully defined atoms or undefined. Thus an array may be partially defined. Indeed, because the arrays are nested, the elemnts of an array may be partial elements. The arrays of different size are not related to each other so the entire domain decomposes into an infinite separated sum. This is possible to formalize using the Σ type-constructor but we can give a much simpler account of the domain.

In defining the abstract semantics of expressions in Cid we shall use closure operators. These will be closure operators on the domain $V \times ENV$ where V is the domain of values and ENV is the domain of environments. Thus, the meaning of an expression will be a function of type $(ENV \times V) \to (ENV \times V)$. The evaluation of an expression adds information about the value of the expression by imposing a constraint on it and may put constraints on other variables as well. These other constraints are expressed via the environment that is returned.

Intuitively one may read these meanings in the following way. Each expression is supplied with an environment and an estimate of the resulting value. The semantics refines the environment to incorporate the effect of any new constraints that may result from the evaluation of the expression and refines the value supplied to incorporate the effect of the expression evaluation. It is important to keep in mind that this does not correspond to the operational semantics it is simply an intuitive way of thinking about the closure operators.

To illustrate these ideas we shall describe a simple example. Consider the equations $x = array(2), x[1] = 1, x[2] = 2$. These may be part of a Cid program. We view these as imposing constraints. The first equation says that x is an array of size 2. The closure operator that represents this can be defined as follows. The array $[\bot, \bot]$ is the least array of size 2, let us call this element $\bot_2$. Now the closure operator is just $\lambda u.u \sqcup \bot_2$. Similarly the closure operators representing the next two constraints are $\lambda v.v \sqcup [1, \bot]$ and $\lambda v.v \sqcup [\bot, 2]$. The composite of these three functions is $\lambda u.u \sqcup [1, 2]$. Clearly the least fixed point of the composite function is $[1, 2]$. A less trivial example is obtained by having $x = array(2), x[1] = x[2]$. The closure operator representing these two constraints is $\lambda u.let\ v = u[1] \sqcup u[2]\ in\ [v, v]$.

To obtain an element of the set of values that satisfy this constraint we supply an approximation, say [a,b], and we will get as result $[a \sqcup b, a \sqcup b]$, which clearly satisfies the constraint.

5.1 The Semantic Domain

The basic data structuring mechanism available in Cid is the array data type. The arrays can be nested and, because evaluation is done in parallel, the array constructor is non-strict. This allows arrays to be nested infinitely deeply. Thus the domain we use cannot be the simple array construct seen in most text-books but has to involve solving recursive domain equations [11]; as is done, for example, with infinite streams. However, one cannot simple adapt the solution used for streams or lazy S-expressions as we shall discuss below.

As we have already indicated, the need for T becomes clear when we consider the meaning of constraints like $x = 2, x = 3$. Mathematically, the necessity of top manifests itself in that very few closure operators can be defined on a domain without a top element. Consider the "usual" domain of integers, $N_\perp$, i.e. the flat domain with a bottom element but with no top element. A closure operator, say f, has to satisfy $x \sqsubseteq f(x)$. Thus, a closure operator on $N_\perp$ has to take each defined integer to itself. On the other hand, f has to be monotonic as well. Thus, if $f(\perp)$ is any value other than $\perp$, say n, then $\forall x \in N_\perp$ we must have $f(x) = n$ which is impossible if f is to be a closure operator. Thus, the only closure operator definable is the identity function from $N_\perp$ to itself.

It is worth understanding why we cannot view arrays as syntactic sugar for S-expressions. It is easy to solve the following recursive domain equation for S-expressions:

$$S = A + S \times S$$

We could think of an n-array as an appropriately nested S-expression. However, we would like to have as a possible constraint on a variable x in Cid the stipulation that it be an array of some fixed size, say k. The least element satisfying this constraint is the k-array with none of its entries defined. Thus we would like to have an element, $\perp_k$ for each k, such that $\perp_k$ is less than any other k-array and for any k and k' such that $k \neq k'$ we have $\perp_k$ and $\perp_{k'}$ have no upper bound other than the greatest element.

This would not be the case is we used nesting of S-expressions to encode arbitrary size arrays.

To define the domain of arrays we use a standard construction for defining a domain of (possibly infinite) terms in logic programming, see, for example, Lloyd [3]. First we need some notation. Let ω be the set of natural numbers. We use ω^* for the set of finite sequences of integers. A sequence is written $[i_1, \ldots, i_n]$. If s and t are sequences then $[s,t]$ denotes their concatenation, if s is a sequence and n is a natural number then $[s,n]$ is the sequence s with n added to the end. The size of a set X is witten $|X|$ and the size of a sequence s is written $|s|$.

Definition 6. 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, is 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 T. 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 7. 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 $dom(f) \subseteq dom(g)$ and $\forall s \in dom(f).br(s, dom(f)) \neq 0 \Rightarrow br(s, dom(g)) = br(s, dom(f)) \wedge br(s, dom(f)) = 0 and f(s) \neq \Omega \Rightarrow g(s) = f(s)$.

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. It is straightforward to check that the domain is algebraic and consistently complete, though we never make use of these properties.

5.2 The Denotational Semantics of Cid

In defining the denotational semantics, we need an environment that assigns to identifiers values in V; we shall typically use env to stand for an environment, the type of env is $Id \to V$. The type of the semantic function $\mathcal{E}$ is

$$Exp \to (ENV \times V) \to_C (ENV \times V)$$

One can understand this as follows. The meaning of an expression is a closure operator that takes an environment and a value and treats that value as an approximation to the value of the expression being computed. A new value is produced, this being the refinement to the value supplied. New constraints on the logic variables are incorporated into the new environment produced. We will usually use the curried form getting a type for $\mathcal{E}$ of

$$Exp \to (ENV \to V) \to_C (ENV \times V)$$

where the subscript on the arrow signifies that we are looking at the domain of closure operators rather than the domain of all continuous functions.

We assume that the environment contains one distinguished binding, namely the one for the single function name that is permitted in a Cid program. Strictly speaking then, the type of ENV is modified so that an environment is the sum of an ordinary environment as described above and afunctional environment. We will not make this explicit in the semantic clauses below except when we define the meaning of the function expression. In a subsequent paper, we plan to discuss the semantics of programs with higher-order functions; for the present paper we are only looking at the first-order case so this rather naive treatment of the functional expressions is not problematic. The point is that the expressions in Cid do not place constraints on the functional expression so we can safely factor out the treatment of this case.

The pair returned is formed with the a special pair constructor, written $\langle , \rangle$, which is strict with respect to T. We use the notation lcs to stand for the *least common solution* of a set of equations. Because the functions used are all extensive one easily show that they have a least common solution. 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 $x = a \sqcup x$. We also assume that the basic operations *op* are strict, with respect to $\perp$ and T.

The semantic clauses are:

$$\mathcal{E}[\![const]\!]\ env\ a = \langle env, \mathcal{K}(const) \sqcup a\rangle \tag{1}$$

$$\mathcal{E}[\![x]\!]\ env\ a = \langle env[x \mapsto (env(x) \sqcup a)], env(x) \sqcup a\rangle \tag{2}$$

$$\mathcal{E}[\![e_1\ op\ e_2]\!]\ env\ a = \mathbf{lcs}\left\{\begin{array}{c} env \sqsubseteq env' \\ env', v_1 = \mathcal{E}[\![e_1]\!]\ env'\ v_1 \\ env', v_2 = \mathcal{E}[\![e_2]\!]\ env'\ v_2 \\ r = (v_1\ op\ v_2) \sqcup a \end{array}\right. \\ \mathbf{in}\ \langle env', r\rangle \tag{3}$$

$$\mathcal{E}[\![array(e)]\!]\ env\ a = \mathbf{lcs}\left\{\begin{array}{c} env \sqsubseteq env' \\ env', n = \mathcal{E}[\![e]\!]\ env'\ n \\ r = \mathcal{A}rray(n) \sqcup a \end{array}\right. \\ \mathbf{in}\ \langle env', r\rangle \tag{4}$$

$$\mathcal{E}[\![[L1, L2, \ldots, Ln]]\!]\ env\ a = \\ \mathbf{lcs}\left\{\begin{array}{c} env \sqsubseteq env' \\ a \sqsubseteq r \\ env'[L1] = r[1] \\ \ldots \\ env'[Ln] = r[n] \\ r = (v_1\ op\ v_2) \sqcup a \end{array}\right. \\ \mathbf{in}\ \langle env', r\rangle \tag{5}$$

$$\mathcal{E}[\![cond(e_1, e_2, e_3)]\!]\ env\ a = \\ \mathbf{lcs}\left\{\begin{array}{c} env \sqsubseteq env' \\ env', b = \mathcal{E}[\![e_1]\!]\ env'\ b \end{array}\right. \\ \mathbf{in}$$

$$
\begin{aligned}
& \qquad if\ b\ then\ \mathcal{E}[\![e_2]\!]\ \ env'\ \ a \\
& \qquad else\mathcal{E}[\![e_3]\!]\ \ env'\ \ a \qquad (6) \\
\mathcal{E}[\![e_1[e_2]]\!]\ \ env\ \ a \ &= \\
& \mathbf{lcs} \begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r \\ env', v_1 = \mathcal{E}[\![e_1]\!]\ \ env'\ \ v_1 \\ env', v_2 = \mathcal{E}[\![e_2]\!]\ \ env'\ \ v_2 \\ v_1[v_2] = r \end{cases} \\
& \mathbf{in}\ \langle env', r\rangle \qquad (7) \\
\mathcal{E}[\![F(e)]\!]\ \ env\ \ a \ &= \\
& \mathbf{lcs} \begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r \\ \langle env', v\rangle = \mathcal{E}[\![e]\!]\ \ env'\ \ v \\ env', v, r = \mathcal{E}_F[\![F]\!]\ \ env'\ \ v\ \ r \end{cases} \\
& \mathbf{in}\ \langle env', r\rangle \qquad (8)
\end{aligned}
$$

The last semantic clause above uses the auxiliary function $\mathcal{E}_F$. This function defines the meaning of functional expressions. We assume that all functions have exactly one argument in order not to clutter up the notation. The function $\mathcal{E}_F$ takes an environment and looks up the definition in the functional part of the environment giving a closure operator on the function space, i.e. an element of $(V \to V) \to_C (V \to V)$. This element is given by the function $\mathcal{F}[\![F]\!]$. This function is defined below using a least fixed point operator, written μ, on the space of closure operators on the function space. We assume that the symbol F is bound to $\mathcal{F}[\![F]\!]$ in all the environments used in the definition of $\mathcal{E}$.

$$
\begin{aligned}
\mathcal{F}[\![F]\!] \ &= \ \mu\, f.\lambda\,(env, v, a). \\
& \mathbf{lcs} \begin{cases} \{x \mapsto v, y \mapsto \perp, F \mapsto f\} \\ \sqsubseteq env' \\ a \sqsubseteq r \\ env \sqsubseteq env' \\ env' = \mathcal{C}[\![def - list]\!]\ \ env' \\ \langle env', r\rangle = \mathcal{E}[\![exp]\!]\ \ env'\ \ r \end{cases} \\
& \mathbf{in}\ \langle env'[x], r\rangle
\end{aligned}
$$

The semantic function $\mathcal{C}$, in the above definition of $\mathcal{F}$, defines the effect of declarations. The declarations take the form of equations, *ide* $=$ *exp*, and are viewed as constraints on the value of *ide*. These constraints are expressed in the environment as follows:

$$
\begin{aligned}
\mathcal{C}[\![x = e]\!]\ env \quad &= \\
&\mathbf{lcs}\begin{cases} env \sqsubseteq env' \\ env', r = \mathcal{E}[\![e]\!]\ env'\ r \\ env'[x] = r \end{cases} \\
&\mathbf{in}\ \mathit{if}\ r = T\ \mathit{then}\ env_T\ \mathit{else}\ env' \\
\mathcal{C}[\![def_1\ ;\ def_2]\!]\ env \quad &= \\
&\mathbf{lcs}\begin{cases} env \sqsubseteq env' \\ env' = \mathcal{C}[\![def_1]\!]\ env' \\ env' = \mathcal{C}[\![def_2]\!]\ env' \end{cases} \\
&\mathbf{in}\ env'
\end{aligned}
$$

The first of the above two equations says that if the constraint is inconsistent then the entire environment becomes inconsistent, in the sense that every identifier is bound to T. Thus the effect of an inconsistency is propagated throughout the computation.

This abstract semantics for Cid expresses the effect of program constructs as closure operators on the domain V. The effect of parallel computations is captured by viewing each of the computations as putting constraints on data values. It is critical that the formalism allows for the simultaneous solution of several fixed point equations; using closure operators allows just this. Of course, the denotational semantics given here needs to be related to the operational semantics in order to verify that these abstract semantic descriptions really do correspond to the execution of Id programs.

6 Relating the Two Semantics

In defining the denotational semantics of a programming language it is important to ensure that the denotational semantics and the operational semantics "correspond" in some appropriate way. Ideally, this correspondence takes the form of a full abstraction theorem [4,7]; i.e. every phrase in the programming language is assigned a meaning by the denotational semantics in such a way that two phrases are given equal meanings iff they have the same operational behavior when inserted in all contexts. Recent work on this problem has indicated that fully abstract models are very difficult to come by [1,12] so one often searches for a less stringent requirement. An example of such a correspondence is Wadsworth's theorem, which shows that, in the D_∞ model of the λ-calculus, terms are assigned $\perp$ iff they fail to terminate under head-reduction.

In our case we cannot simply insist that $\perp$ model non-termination. The presence of parallel evaluation means that we need to express the possibility of a subcomputation returning a value while other subcomputations are still in progress. Furthermore, in our abstract semantics the partial order represents increasingly constrained values, thus $\perp$ ought to represent complete lack of constraint. It is possible that one can have a terminating computation that does not impose any non-trivial constraints; for example, $x = x$ does not impose any constraint and terminates immediately. Thus, in order to get a correspondence between $\perp$ and non-termination one needs another denotational semantics that models "quiesence" of dataflow computations. Such a semantic account is under development, we will not discuss it further here.

The proof of the full abstraction theorem is carried out in three stages. First, we show that a single reduction step preserves meaning. This is a basic soundness result for the denotational semantics. Next, we show that we can always construct a reduction sequence that attains the value specified by the denotational semantics. The proof of this involves the cnstruction of an inclusive predicate to relate semantic values and syntactic expressions. These two results establish the 'adequacy' of the denotational semantics. Finally, we define a suitable operational preorder, as in Berry, Curien and Levy [1], and establish the full abstraction theorem. For this it is important that the closure operators form an algebraic cpo.

6.1 One-step Reduction Preserves Meaning

In this section we 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 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 formalise this notion first.

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 $A1, A2, \ldots$ in 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 possiblity of an inconsistency. If we have an inconsistent alias set A, and $\{x1, \ldots, xN\}$ are all the identifiers in A then $EQ(A) = \{x1 = T, x2 = T, \ldots, xN = T\}$. If we have a consistent alias set then 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 next lemma says that all 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 $\overset{*}{\leadsto}$preserves the meaning.

Lemma 1. If $\rho \overset{*}{\leadsto} \rho'$ then $EQ(\rho) = EQ(\rho')$.

Proof: We know, by theorem 1, that the sequence of $\leadsto$ steps terminates, thus we need only show that if $\rho \leadsto \rho'$ then $EQ(\rho) = EQ(\rho')$. We can now consider the two cases in definition 2. In the first case, the new equations that result from the merging of the two alias sets were already added when we performed the transitive closure of $EQ(\rho)$. In the second case, the equations that result from creating the new alias sets are present when we perform the decomposition of the arrays described in case (iii) above. Thus we generate the same set of equations.

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 $\mathcal{E}, \mathcal{F}$ and $\mathcal{C}$ defined previously and the same notational conventions.

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

We require that the semantic environment env and the syntactic environment ρ satisfy

$$Dom(env) \cap FL = \emptyset \cdots\cdots \star$$

so that there will be no conflicts occurring when the rewriting needed for array allocation is performed. The function $\mathcal{M}$, defines the meaning of expressions in the context of resolved constraints (represented by ρ) as well as equations representing constraints that have not been resolved yet (represented by equations in D). Thus, it 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 not alter. Since $\mathcal{M}$ assigns a closure operator to an operational configuration, this is equivalent to saying that the set of fixed points of the closure operator assigned to an operational configuration is preserved under reduction of the configuration. More precisely, we prove that the part of the environment that is initially relevant is preserved by the one-step reduction. The reason we need this restriction is that some of the rewrites may cause new variables to be generated; in that case one clearly cannot hope that the environments are identical. We use the notation $|_{bv(\rho)}$ to mean that the resulting environment is restricted to the variables that were bound in the environment ρ.

The following theorem states that all the solutions of the constraints are the same before and after a rewrite step provided one ignores new variables that may have been introduced by the rewrite.

Theorem 3. Suppose that the following rewrite is possible:

$$< D, e, \rho, FL > \;\rightarrow\; < D', e', \rho', FL' >$$

then $\forall$ *env* satisfying the condition $\star$ with respect to both ρ and ρ' and $\forall\, a \in V$

$$\mathcal{M}[\![\langle D, x, \rho, FL\rangle]\!] \;\; env \;\; a|_{bv(\rho)} = env \;\; a|_{bv(\rho)} \Longleftrightarrow$$
$$\mathcal{M}[\![\langle D', e', \rho', FL'\rangle]\!] \;\; env \;\; a|_{bv(\rho)} = env \;\; a|_{bv(\rho)}$$

Proof:
The proof proceeds by induction on the size of the proof that the one-step reduction applies. The base cases are the unconditional rewrites.

$$< D, x, \rho, FL > \;\rightarrow\; < D, \rho[x], \rho, FL >$$

Using the definition of $\mathcal{M}$ we get:

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

So, $\mathcal{M}[\![\langle D, x, \rho, FL\rangle]\!] \;\; env \;\; a = \; env \; a$ can be equivalently stated as

$$\begin{array}{rcl}
env' & = & \mathcal{C}[\![D \cup \rho]\!] \;\; env' \\
env', a & = & \mathcal{E}[\![x]\!] \;\; env' \;\; a
\end{array}$$

Assuming that $\rho[x] = e$, our aim is to prove that the above is equivalent to

$$\begin{array}{rcl}
env' & = & \mathcal{C}[\![D \cup \rho]\!] \;\; env' \\
env', a & = & \mathcal{E}[\![e]\!] \;\; env' \;\; a
\end{array}$$

This reduces to proving that

$$\begin{aligned} env' &= \mathcal{C}[\![x = e]\!]\ env' \\ env', a &= \mathcal{E}[\![x]\!]\ \ env'\ \ a \end{aligned}$$

and

$$\begin{aligned} env' &= \mathcal{C}[\![x = e]\!]\ env' \\ env', a &= \mathcal{E}[\![e]\!]\ \ env'\ \ a \end{aligned}$$

are equivalent. Note that $\mathcal{C}[\![x = e]\!]$ env is defined as

$$\begin{array}{l} \textbf{lcs} \left\{ \begin{array}{c} env \sqsubseteq env' \\ env', r = \mathcal{E}[\![e]\!]\ \ env'\ \ r \\ env'[x] = r \end{array} \right. \\ \textbf{in if } r = T \textbf{ then } env_T \textbf{ else } env' \end{array}$$

So $\mathcal{C}[\![x = e]\!]\ env'= env'$ splits up into the following two cases:

1. ($env' \neq env_T$)
   In this case

   $$\begin{aligned} env', r &= \mathcal{E}[\![e]\!]\ \ env'\ \ r \\ env'[x] &= r \end{aligned}$$

   imply that $env', r = \langle env'[x \mapsto env'[x] \bigsqcup r], env'[x] \bigsqcup r \rangle$. Hence, we have $env'[x] \bigsqcup r = r = env'[x]$. Hence, $\langle env', a\rangle$ is a solution of one set of equations if and only if it is so of the other.

2. $(env = env_T)$

   In this case, since the pairing function is strict with respect to T, env_T, we note that $\langle env_T, T\rangle = T$ is a solution of both sets of equations.

The next case we need to consider is the basic binary operation.

Basic Operations: $$\frac{< D, e_1, \rho, FL > \ \rightarrow\ < D^*, e_1^*, \rho^*, FL^* >}{< D, e_1\ op\ e_2, \rho, FL > \ \rightarrow\ < D^*, e_1^*\ op\ e_2, \rho^*, FL^* >}$$

The meaning of the configuration $\mathcal{M}[\![\langle D, e_1 \; op \; e_2, \rho, FL\rangle]\!] \; env \;\; a$ is

$$\mathcal{M}[\![\langle D, e_1 \; op \; e_2, \rho, FL\rangle]\!] \; env \;\; a \;\; =$$

$$\begin{array}{l} \textbf{lcs} \left\{ \begin{array}{cr} env \sqsubseteq env' & 1 \\ a \sqsubseteq r' & 2 \\ env' = \mathcal{C}[\![D \cup \rho]\!] \;\; env' & 3 \\ env', v_1' = \mathcal{E}[\![e_1]\!] \;\; env' \;\; v_1' & 4 \\ env', v_2' = \mathcal{E}[\![e_2]\!] \;\; env' \;\; v_2' & 5 \\ r' = (v_1' \; op \; v_2') \sqcup a & 6 \end{array} \right. \\ \textbf{in} \; \langle env', r'\rangle \end{array}$$

The meaning of the configuration after the rewrite is

$$\mathcal{M}[\![\langle D, e_1^* \; op \; e_2, \rho, FL\rangle]\!] \; env \;\; a \;\; =$$

$$\begin{array}{l} \textbf{lcs} \left\{ \begin{array}{cr} env \sqsubseteq env'' & 7 \\ a \sqsubseteq r'' & 8 \\ env'' = \mathcal{C}[\![D^* \cup \rho^*]\!] \;\; env'' & 9 \\ env'', v_1'' = \mathcal{E}[\![e_1^*]\!] \;\; env'' \;\; v_1'' & 10 \\ env'', v_2'' = \mathcal{E}[\![e_2]\!] \;\; env'' \;\; v_2'' & 11 \\ r'' = (v_1'' \; op \; v_2'') \sqcup a & 12 \end{array} \right. \\ \textbf{in} \; \langle env'', r''\rangle \end{array}$$

The two systems of equations are identical except for the fact that we have D^* instead of D and similarly for e_1 and ρ. The induction hypothesis allows us to conclude that the two sets of equations have the same set of solutions.

The reasoning for the conditional is similar. The only subtlety is that the evaluation does not proceed until the predicate has been fully reduced. This is important because if we were to evaluate both arms of the conditional in parallel before waiting for the result of the boolean evaluation there could be inconsistent constraints imposed on variables and the result of the computation would be T.

The next case we need to consider in detail is the case of the array. The rewrite rule is

$$\text{Array: } 1. \; \frac{< D, e, \rho, FL > \;\; \rightarrow \;\; < D^*, e^*, \rho^*, FL^* >}{< D, array(e), \rho, FL > \;\; \rightarrow \;\; < D^*, array(e^*), \rho^*, FL^* >}$$

$$2. \; < D, array(n), \rho, FL > \;\; \rightarrow \;\; < D, [L1, ..., Ln], \rho^*, FL^* >$$

where L1,...,Ln $\in FL$

$\rho^* = \rho \cup \{\{\mathrm{L1}\},...\{\mathrm{Ln}\}\}$
$FL^* = FL - \{L1, ...Ln\}$

We first consider the first transition. The proof reduces to showing that

$$\begin{aligned} env' &= \mathcal{C}[\![D \cup \rho]\!]\ env' \\ env', n' &= \mathcal{E}[\![e]\!]\ env'\ n' \\ r' &= Array(K(n)) \bigsqcup r' \end{aligned}$$

and

$$\begin{aligned} env'' &= \mathcal{C}[\![D^* \cup \rho^*]\!]\ env'' \\ env'', n'' &= \mathcal{E}[\![e^*]\!]\ env''\ n'' \\ r'' &= Array(K(n)) \bigsqcup r'' \end{aligned}$$

have the same set of solutions. The result follows from the induction hypothesis on the operational transition $< D, e, \rho, FL > \ \rightarrow\ < D^*, e^*, \rho^*, FL^* >$.

The second rewrite rule is applicable when the expression e has reduced to an integer. The meanings of the two configurations are

$$\mathcal{M}[\![\langle D, array(n), \rho, FL\rangle]\!]\ env\ a\ =$$

$$\textbf{lcs} \begin{cases} env \sqsubseteq env' & 1 \\ a \sqsubseteq r' & 2 \\ env' = \mathcal{C}[\![D \cup \rho]\!]\ env' & 3 \\ r' = Array(K(n)) \sqcup a & 4 \end{cases}$$
$$\textbf{in}\ \langle env', r'\rangle$$

and

$$\mathcal{M}[\![\langle D, array(n), \rho, FL\rangle]\!]\ env\ a\ =$$

$$\textbf{lcs} \begin{cases} env \sqsubseteq env'' & 5 \\ a \sqsubseteq r'' & 6 \\ env'' = \mathcal{C}[\![D \cup \rho]\!]\ env'' & 7 \\ env''[L1] = r'[1] & 8 \\ \dots & \\ env''[Ln] = r'[n] & 9 \end{cases}$$
$$\textbf{in}\ \langle env'', r''\rangle$$

In these expressions, the constraints are identical, except for the constraints on r' and r''. In both cases, however, all that the constraints require are that the result be an array of size n with values above those prescribed in a. The new environments env' and env'' will differ in that the former will have no bindings for the identifiers $L1, \ldots, Ln$ but the operational semantics ensures that these are new identifiers, hence the environments env' and env'' will agree on the variables that had been defined before the rewrite occurred. Showing that the rewrite rules for array selection preserve meaning is straightforward.

The final case that we look at is function application. The operational rules are.

Application:

$$< D, F(e), \rho, FL > \ \rightarrow \ < D^*, e_1, \rho^*, FL^* >$$
where $D^* = D \cup \{x = e\} \cup defs_F[x/arg_F][y/local_F](x, y \in FL)$
$e_1 = exp_F[x/arg][y/local_F]$
$\rho^* = \rho^* \cup \{\{x\}, \{y\}\}$
$FL^* = FL - \{x, y\}$

The meanings of the configurations are given by

$$\mathcal{M}[\![\langle D, F(e), \rho, FL\rangle]\!] \ env \ a \ = \ \mathbf{lcs} \left\{ \begin{array}{rr} env \sqsubseteq env' & 1 \\ a \sqsubseteq r' & 2 \\ env' = \mathcal{C}[\![D \cup \rho]\!] \ env' & 3 \\ \langle env', v'\rangle = \mathcal{E}[\![e]\!] \ env' \ v' & 4 \\ \langle v', r'\rangle = \mathcal{F}[\![F]\!]v'r' & 5 \end{array} \right. \ \mathbf{in} \ \langle env', r'\rangle$$

and

$$\mathcal{M}[\![\langle D^*, e_1, \rho^*, FL^*\rangle]\!] \ env \ a \ = \ \mathbf{lcs} \left\{ \begin{array}{rr} env \sqsubseteq env'' & 6 \\ a \sqsubseteq r'' & 7 \\ env'' = \mathcal{C}[\![D^* \cup \rho^*]\!] \ env'' & 8 \\ \langle env'', r''\rangle = \mathcal{E}[\![e_1]\!] \ env'' \ r'' & 9 \end{array} \right. \ \mathbf{in} \ \langle env'', r''\rangle$$

We need to show that as far as the constraints that affect the old variables are concerned, the solutions of the equations

$$\begin{aligned}\mathcal{M}[\![\langle D, F(e), \rho, FL\rangle]\!]\ env\ a &= env, a \\ \mathcal{M}[\![\langle D^*, e_1, \rho^*, FL^*\rangle]\!]\ env'\ r &= env', r\end{aligned}$$

are identical. We need to show that all solutions of 8 and 9 co-incide with solutions of 3,4 and 5 and vice-versa.

Equation 8 contains all the equations implicit in 3 as well as the new ones obtained by adding the definitions in F to the configuration. The constraint on the argument to the function contained in equation 4 is contained in equation 8 because the rewrite rule explicitly puts the equation $x = e$ into D^*. The two systems of equations express the same constraints, thus $\mathcal{M}$ assigns the same meanings to the two configurations.

The final issue that we need to address is the soundness of the rewrite rules that use unification to incorporate new identifiers into the collection of alias sets i.e to show that the cases labeled "definitions" in section defining the operational semantics preserve the meanings of configurations. These are as follows.

Definitions:

1. $$\frac{< D, e, \rho, FL > \rightarrow < D^*, e^*, \rho^*, FL^* >}{< D \cup \{x = e\}, e_1, \rho, FL > \rightarrow < D^* \cup \{x = e^*\}, e_1, \rho^*, FL^* >}$$

2. $< D \cup \{x = y\}, e, \rho, FL > \rightarrow < D, e, \mathcal{U}(\rho, \{x, y\}), FL >$
   (if $\mathcal{U}(\rho, \{x, y\})$ is consistent)

   $< D \cup \{x = y\}, e, \rho, FL > \rightarrow Error$ (otherwise)

3. $< D \cup \{x = c\}, e, \rho, FL > \rightarrow < D, e, \mathcal{U}(\rho, \{x, c\}), FL >$
   (if $\mathcal{U}(\rho, \{x, c\})$ is consistent)

   $< D \cup \{x = c\}, e, \rho, FL > \rightarrow Error$ (otherwise)

4. $< D \cup \{x = [L1, ..., Ln]\}, e, \rho, FL > \rightarrow < D, e, \mathcal{U}(\rho, \{x, [L1, ..., Ln]\}), FL >$
   (if $\mathcal{U}(\rho, \{x, [L1, ..., Ln]\})$ is consistent)

$$< D \cup \{x = [L1, ..., Ln]\}, e, \rho, FL > \rightarrow Error \text{ (otherwise)}$$

The reasoning is quite straightforward now. There are four sub-cases to consider. The first case follows immediately from the inductive hypothesis. In the second case, we add the equation $x = y$ to ρ and thus to $EQ(\rho)$ but it was already present in D thus it was present as a constraint in computing the meaning of the configuration. Similarly, if there is an inconsistency introduced by the unification process then there were inconsistent constriants present in computing the meaning of the original configuration and setting the meaning to T preserves meaning. The third case is exactly like the second. For the fourth case, we note that the new equations generated by unification were present in the combined constraints imposed by D and ρ.

6.2 Adequacy of the Denotational semantics

In this section we prove that the operational semantics actually attains the values predicted by the denotational semantics. This together with the last theorem says that the denotational semantics and the operational semantics match exactly; this is usually called adequacy. Of course we still will have to prove that this correspondence persists in all contexts. Since infinite objects are present in the semantic domain we cannot say that if the denotational semantics predicts a value then that value is actually attained by a *finite* reduction sequence. What we say instead is, roughly speaking, that there is a reduction sequence to every *finite approximant* of the predicted value.

6.2.1 Some operational facts

We first develop the tools that we need for the proof. The proofs in this subsection are quite routine and are omitted.

Definition 8. $\langle D_1, e_1, \rho_1, FL_1\rangle$ and $\langle D_2, e_2, \rho_2, FL_2\rangle$ 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 for renaming.

Lemma 2. If $\langle D_0, e_0, \rho_0, FL_0\rangle \longrightarrow conf_1$ and $\langle D_0, e_0, \rho_0, FL_0\rangle \longrightarrow conf_2$, then

1. If $conf_1 =$ **error** , $conf_2 \longrightarrow$ **error**
2. If $conf_2 =$ **error** , $conf_1 \longrightarrow$ **error**
3. $conf_1 = \langle D_1, e_1, \rho_1, FL_1\rangle$, $conf_2 = \langle D_2, e_2, \rho_2, FL_2\rangle$, and $(FL_0 - FL_1)\bigcap(FL_0 - FL_2) = \emptyset$, then one of the following holds:
   (a) $conf_1 \xrightarrow{\alpha} conf_2$
   (b) $conf_1 \longrightarrow$ **error**, $conf_2 \longrightarrow$**error**
   (c) $\exists\,\langle D_3, e_3, \rho_3, FL_3\rangle$ such that $\langle D_1, e_1, \rho_1, FL_1\rangle \longrightarrow \langle D_3, e_3, \rho_3, FL_3\rangle$ and $\langle D_2, e_2, \rho_2, FL_2\rangle \longrightarrow \langle D_3, e_3, \rho_3, FL_3\rangle$

Proof: The proof is by induction on the length of the proofs of

$$\langle D_0, e_0, \rho_0, FL_0\rangle \longrightarrow conf_1$$

and

$$\langle D_0, e_0, \rho_0, FL_0\rangle \longrightarrow conf_2.$$

The next lemma generalizes the previous lemma to arbitrarily long reduction sequences. It is essentially a Church-Rosser theorem.

Lemma 3. If $\langle D_0, e_0, \rho_0, FL_0\rangle \xrightarrow{*} conf_1$ and $\langle D_0, e_0, \rho_0, FL_0\rangle \xrightarrow{*} conf_2$, then

1. If $conf_1 =$ **error** , $conf_2 \xrightarrow{*}$ **error**
2. If $conf_2 =$ **error** , $conf_1 \xrightarrow{*}$ **error**
3. $conf_1 = \langle D_1, e_1, \rho_1, FL_1\rangle$, $conf_2 = \langle D_2, e_2, \rho_2, FL_2\rangle$, and

$$(FL_0 - FL_1)\bigcap(FL_0 - FL_2) = \emptyset,$$

$$\exists\langle D_3, e_3, \rho_3, FL_3\rangle such that\langle D_1, e_1, \rho_1, FL_1\rangle \xrightarrow{*}$$

$$\langle D_3, e_3, \rho_3, FL_3\rangle and\langle D_2, e_2, \rho_2, FL_2\rangle \xrightarrow{*} \langle D_3, e_3, \rho_3, FL_3\rangle$$

The proof is by induction on the length of reduction sequences.

We now introduce an operational notion of approximation between environments.

Definition 9. ρ_1 is an *APPROXIMATION* to ρ_2 if $EQ(\rho_1) \subseteq EQ(\rho_2)$

Lemma 4. $\langle D_0, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_1, e_1, \rho_1, FL_1\rangle \Longrightarrow \rho_0$ is an approximation to ρ_1

The proof is by examining the definition of the reduction relation.

The following lemma states that parallel reductions do not interfere. This can be viewed as a sort of "monotonicity" property; adding constraints does not disable a reduction that is enabled.

Lemma 5. If $\langle D_0, e_0, \rho_0, FL_0\rangle \xrightarrow{*} conf_1$, then

1. If $conf_1 = \textbf{error}$, then

$$\langle D_0 \bigcup D, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \textbf{error}$$

2. If $conf_1 = \langle D_1, e_1, \rho_1, FL_1\rangle$, then

$$\langle D_0 \bigcup D, e_0, \rho_0, FL_0\rangle$$
$$\xrightarrow{*} \langle D_1 \bigcup D, e_1, \rho_1, FL_1\rangle$$

Proof: Induction on the number of one step reductions.

The next two lemmas state that the equations generated from the constraints are consistent with the reduction rules.

Lemma 6. If $x = y \in EQ(D_0 \bigcup \rho_0)$ and $\langle D_0, e_0, \rho_0, FL_0\rangle \xrightarrow{*} conf_1$, and D_0', e_0' are got from D_0, e_0 by replacing all occurrences of y by x, and $FL_0' = FL_0 \bigcup \{y\}$, and $\rho_0' = U(\rho_0, \{x, y\})$ from which all occurrences of y have been removed, then

1. $conf_1 = \textbf{error}$, then $\langle D_0', e_0', \rho_0', FL_0'\rangle \xrightarrow{*} \textbf{error}$

2. $conf_1 = \langle D_1, e_1, \rho_1, FL_1\rangle$, then

   - $\langle D_0', e_0', \rho_0', FL_0'\rangle \xrightarrow{*} \textbf{error}$, or

- $\langle \acute{D}_0, \acute{e}_0, \acute{\rho}_0, F\acute{L}_0 \rangle \xrightarrow{*} \langle \acute{D}_1, \acute{e}_1, \acute{\rho}_1, F\acute{L}_1 \rangle$, where $\acute{D}_1$, $\acute{e}_1$ are got from D_1, e_1 by replacing all occurrences of y by x, and $F\acute{L}_1 = FL_1 \bigcup \{y\}$, and $\acute{\rho}_1 = U(\rho_1, \{x, y\})$ from which all occurrences of y have been removed.

The final lemma in this section states that if a reduction can be carried out in a certain syntactic environment it can also be carried out in an environment that defines more equations modulo possible renaming of finitely many variables.

Lemma 7. If $\langle D_0, e_0, \rho_0, FL_0 \rangle \xrightarrow{*} \langle D_1, e_1, \rho_1, FL_1 \rangle$, and $EQ(\rho_1)$ is an approximation to $EQ(\rho_2)$, and $FL_2 \subseteq FL_1$, then

$\langle D_0, e_0, \rho_2, FL_2 \rangle \xrightarrow{*} \langle \acute{D}_1, \acute{e}_1, \acute{\rho}_1, F\acute{L}_1 \rangle$, where there is a finite set of equations $E = \{x_i = y_i | i = 1 \ldots n\}$, such that

- $\{x_i | i = 1 \ldots n\} = FL_1 - FL_0$
- $\{y_i | i = 1 \ldots n\} = F\acute{L}_1 - FL_2$
- Replacing y_i by x_i in $\acute{D}_1$, $\acute{e}_1$ gives D_1, e_1, and

$$\rho_2 = U(\acute{\rho}_1, \{\{x_1, y_1\} \ldots \{x_n, y_n\}\})$$

  from which all occurrences of $y_1 \ldots y_n$ have been removed.

6.2.2 The Inclusive Predicate

Inclusive predicates are key components in many adequacy proofs [5,13]. They relate the semantic values with syntactic expressions. They are primarily used to establish that a predicted sematic value is actually attained by rewriting. For defining the inclusive predicate, we need to develop notation that relates syntactic and semantic environments.

The following definitions capture the notion of an expression dominating a value or of a syntactic environment dominating a semantic environment. They are defined in terms of configurations that have no unresolved constraints. They are needed for the definition of the inclusive predicate, which is defined for general configurations, that appears in the third definition below.

Definition 10. ρ_0 *COVERS* env_0, if $(\forall x)$

1. $env_0\ [x] = b$, where b is a basic value, $\Longrightarrow$

$$\langle \emptyset, e, \rho, FL\rangle \longrightarrow \langle \emptyset, b, \rho, FL\rangle$$

2. $env_0\ [x] = A$, where A is an array, $\Longrightarrow$
   $(env_0\ [A\langle s\rangle] = b) \Longrightarrow (\langle \emptyset, x\langle s\rangle, \rho_0, FL_0\rangle \xrightarrow{*} \langle \emptyset, b, \rho_0, FL_0\rangle)$
   where b is a basic value, and s is any finite sequence.

Definition 11. Let e be an expression and r a semantic value. We say that e *COVERS* r *IN* ρ_0, if

1. $r = b$, where b is a basic value, $\Longrightarrow$
   $(\langle \emptyset, e, \rho_0, FL_0\rangle \xrightarrow{*} \langle \emptyset, \mathrm{b}, \rho_0, FL_0\rangle)$

2. $r = A$, where A is of type array, $\Longrightarrow$
   $(A\langle s\rangle = b) \Longrightarrow (\langle \emptyset, e\langle s\rangle, \rho_0, FL_0\rangle \xrightarrow{*} \langle \emptyset, b, \rho_0, FL_0\rangle)$
   where b is a basic value, and s is any finite sequence.

Now, we define the inclusive predicate.

Definition 12. $F_0 \preceq e_0$ if when ρ_0 *COVERS* env_0 and $F_0\ env_0 \perp = x$.

1. $x = \top \Longrightarrow \langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*}$ **error**

2. $x = \langle r_1,\ env_1\rangle,\ r_1 \neq \top,\ env_1 \neq env_\top \Longrightarrow$
   $(\langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \textbf{error}) \vee$
   $(\forall(\langle r_{fin},\ env_{fin}\rangle \sqsubseteq \langle r_1,\ env_1\rangle) \Longrightarrow$
   $(\exists\ \langle D_1, e_1, \rho_1, FL_1\rangle\colon \langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_1, e_1, \rho_1, FL_1\rangle)$
   $\wedge\ (\rho_1\ \mathit{COVERS}\ env_{fin}) \wedge (e_1\ \mathit{COVERS}\ r_{fin}\ \mathit{IN}\ \rho_1))$

Roughly speaking, $x \preceq e$ means that when e is evaluated the resulting expression has a meaning that dominates the semantic value x. The significance of the inclusive predicate is that adequacy is the statement that the *meaning* of an expression is related to the expression by $\preceq$.

6.2.3 Proof of adequacy

First, we state a couple of technical results that are used in the proof, and whose proof follows from the definition of the inclusive predicate immediately.

Lemma 8. If ρ_0 is an *APPROXIMATION* to ρ_1 and ρ_0 *COVERS* *env*, then ρ_1 *COVERS* *env*.

Lemma 9. If
e_0 *COVERS* r in ρ_0, and $\langle D_0, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_1, e_1, \rho_1, FL_1\rangle$, then e_1 *COVERS* r in ρ_1.

The difficult aspect of an adequacy 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. In some sense, this is the key to the whole adequacy proof. Suppose that f and g are two closure operators that correspond to the imposition of two constraints. Suppose that we know how to construct reduction sequences corresponding to the resolution of each these constraints individually. Then, because we know that the least commmon fixed-point of f and g is the least fixed-point of $f \circ g$, we can construct an interleaved reduction sequence that corresponds to computing the iterates of $f \circ g$. In other words, the special form of the fixed-point iteration provides guidance about how one can construct the interleaved reduction sequence. The following lemma formalizes this intuition and is given below in detail.

Lemma 10. If $F_0' \preceq e_0$, $F_1' \preceq e_1$, then $F \preceq (e_0 \; op \; e_1)$, where

$$F \; env \; a = \text{lcs} \begin{cases} \acute{env}\; v_0 = & F_0 \; \acute{env} \; v_0 \\ \acute{env}\; v_1 = & F_1 \; \acute{env} \; v_1 \\ & a \sqsubseteq r \\ & v_0 \; op \; v_1 \sqsubseteq r \\ & env \sqsubseteq \acute{env} \end{cases}$$

$$\textbf{in } \langle \acute{env},\; r\rangle$$

Proof: Since there are no other constraints on r, we have,

$$F\ env\ \bot = \textbf{lcs} \begin{cases} \acute{env}\ v_0 = & F_0\ \acute{env}\ \bot \\ \acute{env}\ v_1 = & F_1\ \acute{env}\ \bot \\ & env \sqsubseteq \acute{env} \end{cases}$$

$$\textbf{in}\ \langle \acute{env},\ v_0\ op\ v_1 \rangle$$

Define

$$\begin{aligned} F_0\langle env, v_0, v_1, v\rangle &= \langle env', v_0', v_1, v\rangle \\ F_1\langle env, v_0, v_1, v\rangle &= \langle env', v_0, v_1', v\rangle \\ F_2\langle env, v_0, v_1, v\rangle &= \langle env, v_0, v_1, v \bigsqcup (v_0\ op\ v_1)\rangle \end{aligned}$$

where

$$\begin{aligned} F_0'\ env\ v_0 &= env'\ v_0' \\ F_1'\ env\ v_1 &= env'\ v_1' \end{aligned}$$

These functions F_0, F_1, F_2 each represent the stages in computing F in a particular interleaved computation.

Let pr be the projection function of a 4-tuple on the first and last (fourth) arguments. Note that $F\ env\ a =$

$$pr(\textbf{lcs}\ [F_0\langle env, \bot, \bot, a\rangle,\ F_1\langle env, \bot, \bot, a\rangle, F_2\langle env, \bot, \bot, a\rangle]).$$

So, in particular, we have

$$F\ env\ \bot = pr(\textbf{lcs}\ [F_0\langle env, \bot, \bot, \bot\rangle,\ F_1\langle env, \bot, \bot, \bot\rangle, F_2\langle env, \bot, \bot, \bot\rangle]).$$

Let ρ cover env. Let v_0, v_1, env_f be finite.
We shall prove by induction on k that

Proposition 1. 1. $\langle env_f, v_0, v_1, v_0\ op\ v_1\rangle \sqsubseteq (F_2 \circ F_1 \circ F_0)^k \langle env, \bot, \bot, \bot\rangle \Rightarrow$

(a) $\langle \Phi, e_0\ op\ e_1, \rho, FL\rangle \xrightarrow{*} \textbf{error}$, or

(b) For $j = 1 \ldots 2 \times k$, we have reductions, $\langle D_j, e_{0j}\ op\ e_{1j}, \rho_j, FL_j\rangle \xrightarrow{*} \langle D_{(j+1)}, e_{0(j+1)}\ op\ e_{1(j+1)}, \rho_{(j+1)}, FL_{(j+1)}\rangle$, where

- $D_0 = \phi$, $e_{00} = e_0$, $e_{10} = e_1$, $\rho_0 = \rho$
- If j is even, $e_{1j} = e_{1(j+1)}$
- If j is odd, $e_{0j} = e_{0(j+1)}$
- $e_{0(2\times k)}$, $e_{1(2\times k)}$ cover v_0, v_1 in $\rho_{2\times k}$
- $\rho_{2\times k}$ covers env_f

2. $T \sqsubseteq (F_2 \circ F_1 \circ F_0)^k \langle env, \bot, \bot, \bot\rangle \Rightarrow$
   $\langle \Phi, e_0\ op\ e_1, \rho, FL\rangle \xrightarrow{*}$ **error**

This proposition describes exactly the interleaved reduction sequences for the two subexpressions of $e_0 op e_1$. We first give a proof of the lemma assuming the above.
Since T is a finite element,
$T \sqsubseteq (F_2 \circ F_1 \circ F_0)^k \langle env, \bot, \bot, \bot\rangle \Rightarrow$
$\langle \Phi, e_0\ op\ e_1, \rho, FL\rangle \xrightarrow{*}$ **error**.
Otherwise let env_f, $v_f \sqsubseteq F\ env$, $\bot$. Hence, there are k, v_0, v_1, all finite, such that $\langle env, v_0, v_1, v_0\ op\ v_1\rangle \sqsubseteq (F_2 \circ F_1 \circ F_0)^k \langle env, \bot, \bot, \bot\rangle$, where $v_f \sqsubseteq v_0\ op\ v_1$. If we do not have a reduction to error, $\langle \Phi, e_0\ op\ e_1, \rho, FL\rangle \xrightarrow{*}$ an operational configuration $\langle D_{(2\times k)}, e_{0(2\times k)}\ op\ e_{1(2\times k)}, \rho_{(2\times k)}, FL_{(2\times k)}\rangle$, such that

- $\rho_{(2\times k)}$ covers env_f.
- If $v_f = \bot$, the proof is complete. If not, since *op* is strict, both v_0, v_1 are base values(integers). Since v_0, v_1 are covered by $e_{0(2\times k)}$, $e_{1(2\times k)}$ respectively in $\rho_{(2\times k)}$, result follows by application of the reduction rule that reduces the operator symbol and lemma 9.

Now, we prove the first half of the proposition. The proof of the second part is analagous and is omitted.

- The base case, when $k = 1$. First recall that if a continuous function, when applied to an argument a, produces a *finite* output then some finite approximation of a suffices to produce the same output. Now, from the continuity of *op*, F_0, F_1, F_2 and from the explicit form of their definitions, we deduce the existence of v_0, v_1, both finite, such that

$$\begin{aligned}
\langle env_f, v_0, v_1, v_f\rangle &\sqsubseteq F_2\langle env_f, v_0, v_1, \bot\rangle \\
\langle env_f, v_0, v_1, \bot\rangle &\sqsubseteq F_1\langle env_1, v_0, \bot, \bot\rangle \\
\langle env_1, v_0, \bot, \bot\rangle &\sqsubseteq F_0\langle env, \bot, \bot, \bot\rangle
\end{aligned}$$

The result now follows from the hypothesis on F_0', F_1', by first performing the reductions on e_0, followed by the reductions on e_1 and finally using the reduction rule for *op*.

- The inductive case is no harder in principle but there are tedious details that need to be checked. The idea is exactly the same, we interleave the computations of the subexpressions with the reductions of *op* to produce the required reduction sequence. Assume the result for some $k > 0$. Let ρ cover *env*. Let $\langle env_{f0}, v_0, v_1, v_0\ op\ v_1\rangle \sqsubseteq (F_2 \circ F_1 \circ F_0)^{(k+1)}\langle env, \perp, \perp, \perp\rangle$. From continuity of *op*, F_0, F_1, F_2 and their definitions, we deduce the existence of env', env'', v_0', v_1' such that

  - env', env'', env''', v_0', v_1', are all finite.
  - $\langle env_{f0}, v_0, v_1, v_0\ op\ v_1\rangle \sqsubseteq F_2\langle env_{f0}, v_0, v_1, \perp\rangle$
  - $\langle env_{f0}, v_0, v_1, \perp\rangle \sqsubseteq F_1\langle env', v_0, v_1', \perp\rangle$
  - $\langle env', v_0, v_1', \perp\rangle \sqsubseteq F_0\langle env'', v_0', v_1', \perp\rangle$
  - $\langle env'', v_0', v_1', \perp\rangle \sqsubseteq (F_2 \circ F_1 \circ F_0)^k\langle env, \perp, \perp, \perp\rangle$

  First, we reason that v_0', v_1' can be replaced by $\perp$. Let $v_0' \neq \perp$. Note that the second coordinate of the 4-tuple is altered only by F_0. If $v_0' \neq \perp$, let $j < k$ be the least such that $\langle \perp, v_0, \perp, \perp\rangle \sqsubseteq F_0 \circ (F_2 \circ F_1 \circ F_0)^j\langle env, \perp, \perp, \perp\rangle$. Let $\langle env^*, v_0^*, v_1^*, v^*\rangle = (F_2 \circ F_1 \circ F_0)^k\langle env, \perp, \perp, \perp\rangle$. It follows from the continuity of F_0 that there exists finite $env_{f1} \sqsubseteq env^*$ such that $\langle \perp, v_0, \perp, \perp\rangle \sqsubseteq F_0\langle env_{f1}, \perp, \perp, \perp\rangle$. Similar reasoning can be adopted for v_1' also. Thus, by choosing of $env_f = \bigsqcup[env_{f0},\ env_{f1},\ env_{f2}]$, we deduce the existence of env', env'' such that

  - env', env'' are all finite.
  - $\langle env_f, v_0, v_1, v_0\ op\ v_1\rangle \sqsubseteq F_2\langle env_f, v_0, v_1, \perp\rangle$
  - $\langle env_f, v_0, v_1, \perp\rangle \sqsubseteq F_1\langle env', v_0, \perp, \perp\rangle$
  - $\langle env', v_0, \perp, \perp\rangle \sqsubseteq F_0\langle env'', \perp, \perp, \perp\rangle$
  - $\langle env'', \perp, \perp, \perp\rangle \sqsubseteq (F_2 \circ F_1 \circ F_0)^k\langle env, \perp, \perp, \perp\rangle$

  From the induction hypothesis, either

- $\langle \Phi, e_0\ op\ e_1, \rho, FL\rangle \xrightarrow{*}$ **error**, or
- $\langle \Phi, e_0\ op\ e_1, \rho, FL\rangle \xrightarrow{*} \langle D', e_0'\ op\ e_1', \rho', FL'\rangle$ such that, ρ' covers env''.

In the first case, we are done. For the latter case, consider

$$\langle env', v_0, \bot, \bot\rangle \sqsubseteq F_0\langle env'', \bot, \bot, \bot\rangle.$$

From the hypothesis on F_0', e_0,

- $\langle \Phi, e_0, \rho', FL\rangle \xrightarrow{*}$ **error**, or
- $\langle \Phi, e_0, \rho', FL\rangle \xrightarrow{*} \langle D'', e_0'', \rho'', FL''\rangle$ such that, ρ'' covers env', and e_0'' covers v_0 in ρ''.

Note that we need e_0' instead of e_0 in the left hand side of the above. This is achieved through the use of the lemmae proved on the operational semantics. From finitely many applications of Lemma 7 , $(\exists\langle D^*, e_0^*, \rho^*, FL^*\rangle)$ such that $\langle D'', e_0'', \rho'', FL''\rangle \xrightarrow{*} \langle D^{**}, e_0^{**}, \rho^{**}, FL^{**}\rangle$ where there is a finite set of equations $E = \{x_i = y_i | i = 1 \ldots n\}$, such that Replacing y_i by x_i in D^{**}, e_0^{**} gives D', e_0', and

$$\rho' = U(\rho^{**}, \{\{x_1, y_1\} \ldots \{x_n, y_n\}\})$$

from which all occurrences of $y_1 \ldots y_n$ have been removed. Using the diamond property, we deduce that one of the following holds:

- $\langle D^{**} \bigcup E, e_0^{**}, \rho^{**}, FL^{**}\rangle \xrightarrow{*}$ **error**
- $\langle D^{**} \bigcup E, e_0^{**}, \rho^{**}, FL^{**}\rangle \xrightarrow{*} \langle D''', e_0''', \rho''', FL'''\rangle$ and $\langle D'', e_0'', \rho'', FL''\rangle \xrightarrow{*} \langle D''', e_0''', \rho''', FL'''\rangle$.

Using Lemma 6, we get,

- $\langle D', e_0', \rho', FL\rangle \xrightarrow{*}$ **error**, or
- $\langle D', e_0', \rho', FL\rangle \xrightarrow{*} \langle D'', e_0'', \rho'', FL''\rangle$ such that, ρ'' covers env', and e_0'' covers v_0 in ρ''.

Exactly similar reasoning used on $\langle env_f, v_0, v_1, \bot\rangle \sqsubseteq F_1\langle env', v_0, \bot, \bot\rangle$ gives us the $(2 \times (k+2))th$ set of reductions, which in this case correspond to reductions on the right argument of *op*.

Next, we consider the conditional. This is rather similar to the case of the binary operator so we only sketch the proof.

Lemma 11. If $F_0 \preceq e_0$, $F_1 \preceq e_1$, $F_2 \preceq e_2$, then $F \preceq cond\ (e_o,\ e_1,\ e_2)$, where

$$F\ env\ a = \mathbf{lcs} \left\{ \begin{array}{l} \acute{env}\ b = \ F_0\ \acute{env}\ b \\ \quad env \sqsubseteq \acute{env} \end{array} \right. \\ \mathbf{in}\ if\ b\ then\ F_1\ \acute{env}\ a\ else F_2\ \acute{env}\ a$$

Proof: Since there are no other constraints on b, we have

$$F\ env\ \bot = \mathbf{lcs} \left\{ \begin{array}{l} \acute{env}\ b = \ F_0\ \acute{env}\ \bot \\ \quad env \sqsubseteq \acute{env} \end{array} \right. \\ \mathbf{in}\ if\ b\ then\ F_1\ \acute{env}\ \bot\ else F_2\ \acute{env}\ \bot$$

Let $\rho_0\ COVER\ env$. Let $\langle env_f,\ r\rangle \sqsubseteq F\ env\ \bot$. We have the following cases:

1. $(b =\ true)$

   From continuity of all functions involved and algebraicity of $V \times ENV$, $(\exists env_1)\ (env_1\ : finite)\ (env_1 \sqsubseteq \acute{env})$ such that

$$\begin{array}{rcl} env_f\ ,\ r & \sqsubseteq & F_1\ env_1\ \bot \\ env_1\ ,\ true & \sqsubseteq & F_0\ env\ \bot \end{array}$$

   From hypothesis, one of the following holds:

   (a) $(env_1 = env_\top) \Longrightarrow \langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*}$ **error**

   (b) $((\langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*}$**error**$) \vee$
   $((\exists\ \langle D_3, e_0^\star, \rho_3, FL_3\rangle)$: $\langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_3, e_0^\star, \rho_3, FL_3\rangle)$
   $\wedge\ (\rho_3\ COVERS\ env_1) \wedge e_0^\star\ COVERS\ true\ IN\ \rho_3$.

   - $\langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*}$ **error**
     $\Longrightarrow \langle \emptyset, cond\ (e_0\ , e_1\ , e_2), \rho_0, FL_0\rangle \xrightarrow{*}$ **error**
   - If $((\exists\ \langle D_3, e_0^\star, \rho_3, FL_3\rangle)$: $\langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_3, true, \rho_3, FL_3\rangle 00)$
     $\Longrightarrow \langle \emptyset, cond\ (e_0\ , e_1\ , e_2), \rho_0, FL_0\rangle \xrightarrow{*} \langle D_3, e_1, \rho_3, FL_3\rangle)$
     and the result follows from hypothesis that $F_1 \preceq e_1$

2. ($b =$ *false*). Identical to the above, with F_2 used in the place of F_1.

3. ($b = \top$). $F_0 \preceq e_0$
$\Longrightarrow \langle \emptyset, e_0, \rho_0, FL_0 \rangle \xrightarrow{*}$ **error**
$\Longrightarrow \langle \emptyset, cond\,(e_0\ , e_1\ , e_2), \rho_0, FL_0 \rangle \xrightarrow{*}$ **error**

4. ($b = \perp$) . Result follows from hypothesis $F_0 \preceq e_0$

The array constructor is considered in the following lemma.

Lemma 12. If $F_0 \preceq e_0$, then $F \preceq array\,(e_o)$, where

$$F\ env\ a = \textbf{lcs} \left\{ \begin{array}{c} \acute{env}\ n = F_0\ \acute{env}\ n \\ a \sqsubseteq r \\ array\ (n) \sqsubseteq r \\ env \sqsubseteq \acute{env} \end{array} \right. \\ \textbf{in}\ \langle \acute{env},\ \mathrm{r} \rangle$$

Proof: Since there are no other constraints on n, we have,

$$F\ env\ \perp = \textbf{lcs} \left\{ \begin{array}{c} \acute{env}\ n = F_0\ \acute{env}\ \perp \\ array\ (n) \ = r \\ env \sqsubseteq \acute{env} \end{array} \right. \\ \textbf{in}\ \langle \acute{env},\ \mathrm{r} \rangle$$

Let ρ_0 *COVER* env. Let $\langle env_f,\ r_f \rangle \sqsubseteq \langle env,\ r \rangle$. We have the following cases:

1. ($r_f = \top$)
$\Longrightarrow (n = \top)$
$\Longrightarrow \langle \emptyset, e_0, \rho_0, FL_0 \rangle \xrightarrow{*}$ **error** (as $F_0 \preceq e_0$)
$\Longrightarrow \langle \emptyset, array\,(e_0), \rho_0, FL_0 \rangle \xrightarrow{*}$ **error**

2. ($r_f = \perp$)
$\Longrightarrow (n = \perp)$
Since $F_0 \preceq e_0$, one of the following holds:

   - $(env_f = env_\top) \Longrightarrow \langle \emptyset, e_0, \rho_0, FL_0 \rangle \xrightarrow{*}$ **error**

- $(\langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \mathbf{error}) \vee$
  $(\exists \langle D_1, e_1, \rho_1, FL_1\rangle$: $\langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_1, e_1, \rho_1, FL_1\rangle)$
  $\wedge\ (\rho_1\ COVERS\ env_f)$

So, we have

- $(env_f = env_\top) \Longrightarrow \langle \emptyset, array\ (e_0), \rho_0, FL_0\rangle \xrightarrow{*} \mathbf{error}$
-

$$(\langle \emptyset, array\ (e_0), \rho_0, FL_0\rangle \xrightarrow{*} \mathbf{error}) \vee$$

$$(\exists \langle D_1, e_1, \rho_1, FL_1\rangle : \langle \emptyset, array\ (e_0), \rho_0, FL_0\rangle \xrightarrow{*}$$

$$\langle D_1, array\ (e_1), \rho_1, FL_1\rangle) \wedge (\rho_1 COVERS env_f)$$

3. $(r_f \neq \bot \wedge r_f \neq \top)$
   $\langle env_f,\ n\rangle \sqsubseteq \langle e\acute{n}v,\ n\rangle$ and is *finite*. Since, $F_0 \preceq e_0$, one of the following holds:

   - $(env_f = env_\top) \Longrightarrow \langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \mathbf{error}$
   - $(\langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \mathbf{error}) \vee$
     $(\exists \langle D_1, e_1, \rho_1, FL_1\rangle$: $\langle \emptyset, e_0, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_1, e_1, \rho_1, FL_1\rangle)$
     $\wedge\ (\rho_1\ COVERS\ env_f) \wedge (e_0\ COVERS\ n\ IN\ \rho_1)$

   So, we have

   - $(env_f = env_\top) \Longrightarrow \langle \emptyset, array\ (e_0), \rho_0, FL_0\rangle \xrightarrow{*} \mathbf{error}$
   - $(\langle \emptyset, array\ (e_0), \rho_0, FL_0\rangle \xrightarrow{*} \mathbf{error})$, or
   - $(\exists \langle D_1, e_1, \rho_1, FL_1\rangle$: $\langle \emptyset, array\ (e_0), \rho_0, FL_0\rangle \xrightarrow{*} \langle D_1, array\ (e_1), \rho_1, FL_1\rangle)$

     $\wedge\ (\rho_1\ COVERS\ env_f) \wedge (e_0\ COVERS\ n\ IN\ \rho_1)$
     $\Longrightarrow (\exists \langle D_1, e_1, \rho_1, FL_1\rangle)$ such that
     $\langle \emptyset, array\ (e_0), \rho_0, FL_0\rangle \xrightarrow{*} \langle D_1,\ [L1 \ldots Ln],\ \rho_1 \bigcup \{\{L1\} \ldots \{Ln\},\ FL_1 - \{L1 \ldots Ln\}\rangle$
     $\wedge\ (\rho_1\ COVERS\ env_f)$

The final case is that of function application. It is done by fixed-point induction. We need the folowing two facts about closure operators. The proofs are easy and are omitted.

Lemma 13. Suppose that u is the lcs of the equations

$$\begin{aligned} x &= f(x) \\ x &= g(x) \end{aligned}$$

where $g(x)$ is computed as the lcs of the equations

$$\begin{aligned} x &\sqsubseteq x_1 \\ (x_1, y) &= h(x_1, y) \\ (x_1, y) &= k(x_1, y) \end{aligned}$$

with x_1 returned as the result. Then u can be computed as the lcs of all the equations taken together.

Lemma 14. Suppose u is computed as the lcs of the equations

$$\begin{aligned} x &= f(x) \\ x &= g(x) \end{aligned}$$

where $g = \bigsqcup_i g_i$. Then,

$$u = \bigsqcup_i \text{lcs} \left\{ \begin{array}{l} x = f(x) \\ x = g_i(x) \end{array} \right.$$

Lemma 15. $(\forall e^\star \text{a subexpression of} F\ (e_0) : \mathcal{E}[e^\star] \preceq e^\star) \implies \mathcal{E}[F(e_0)] \preceq F(e_0)$

Proof: We use the notation $a < b$ to mean a is a subexpression of b.

$$\mathcal{E}[F(e_0)]\ env\ a = \text{lcs} \left\{ \begin{array}{c} \acute{env}\ v = \mathcal{E}[e_0]\ \acute{env}\ v \\ \acute{env}\ v\ r = \mathcal{E}_F[F]\ \acute{env}\ v\ r \\ a \sqsubseteq r \\ env \sqsubseteq \acute{env} \end{array} \right.$$

$$\textbf{in}\ \langle \acute{env},\ r \rangle$$

where

$$\mathcal{E}_F[F] = \mu f.\ \lambda\ (env,\ v,\ a)\ \textbf{lcs} \left\{ \begin{array}{c} \{x_0 \mapsto v,\ y_1 \mapsto \perp, \ldots,\ y_n \mapsto \perp,\ F \mapsto f\} \sqsubseteq e\acute{n}v \\ e\acute{n}v =\ C[def - list]\ e\acute{n}v \\ e\acute{n}v\ r =\ \mathcal{E}[exp]\ e\acute{n}v\ r \\ a \sqsubseteq r \\ env \sqsubseteq e\acute{n}v \end{array} \right.$$

$$\textbf{in}\ \langle e\acute{n}v,\ v,\ r\rangle$$

So, we get,

$$\mathcal{E}[F(e_0)]\ env\ a = \textbf{lcs} \left\{ \begin{array}{c} e\acute{n}v\ v =\ \mathcal{E}[e_0]\ e\acute{n}v\ v \\ e\acute{n}v\ v\ r =\ \bigsqcup_i \mathcal{E}_F[F_i]\ e\acute{n}v\ v\ r \\ a \sqsubseteq r \\ env \sqsubseteq e\acute{n}v \end{array} \right.$$

$$\textbf{in}\ \langle e\acute{n}v,\ r\rangle$$

where

1. $\mathcal{E}_F[F_0]\ =\ I$
2. $$\mathcal{E}_F[F_{(i+1)}]\ =\ \lambda\ (env,\ v,\ a)\ \textbf{lcs} \left\{ \begin{array}{c} \{x_0 \mapsto v,\ y_1 \mapsto \perp, \ldots,\ y_n \mapsto \perp,\ F \mapsto \mathcal{E}_F[F_i]\} \sqsubseteq en\acute{v}_{i+1} \\ en\acute{v}_{i+1} =\ C[def - list]\ en\acute{v}_{i+1} \\ en\acute{v}_{i+1}\ r_{i+1} =\ \mathcal{E}[exp]\ en\acute{v}_{i+1}\ r_{i+1} \\ a \sqsubseteq r_{i+1} \\ env \sqsubseteq en\acute{v}_{i+1} \end{array} \right.$$

$$\textbf{in}\ \langle en\acute{v}_{i+1},\ v,\ r_{i+1}\rangle$$

From Lemma 14 ,

$$\mathcal{E}[F(e_0)]\ env\ a = \bigsqcup_i \textbf{lcs} \left\{ \begin{array}{c} e\acute{n}v_i\ v_i =\ \mathcal{E}[e_0]\ e\acute{n}v_i\ v_i \\ e\acute{n}v_i\ v_i\ r_i =\ \mathcal{E}_F[F_i]\ e\acute{n}v_i\ v_i\ r_i \\ a \sqsubseteq r_i \\ env \sqsubseteq e\acute{n}v_i \end{array} \right.$$

$$\textbf{in}\ \langle e\acute{n}v_i,\ r_i\rangle$$

Let

$$\tau_i[E]\ env\ a = \textbf{lcs} \left\{ \begin{array}{c} \acute{env}_i\ v_i =\ \mathcal{E}[E]\ \acute{env}_i\ v_i \\ \acute{env}_i\ v_i\ r_i =\ \mathcal{E}_F[F_i]\ \acute{env}_i\ v_i\ r_i \\ a \sqsubseteq r_i \\ env \sqsubseteq \acute{env}_i \end{array} \right.$$

$$\textbf{in}\ \langle \acute{env}_i,\ r_i \rangle$$

Note that $\mathcal{E}[F(e_0)]\ env\ a\ =\ \bigsqcup_i \tau_i[e_0]\ env\ a$. We shall prove by induction on i that $E < F(e_0) \Longrightarrow \tau_i[E] \preceq F(E)$. This will complete the proof as

$\langle env_f,\ r_f \rangle \sqsubseteq \bigsqcup_i \tau_i[e_0]\ env\ \bot \Longrightarrow (\exists i)\langle env_f,\ r_f \rangle \sqsubseteq \tau_i[e_0]\ env\ \bot$, and we have $\tau_i[e_0] \preceq F(e_0)$

- (Base: i = 0).
  Let ρ_0 *COVER* env. Let $x = \langle env_f,\ r_f \rangle$.

$$\tau_0[E]\ env\ \bot = \textbf{lcs} \left\{ \begin{array}{c} \acute{env}_0\ v_0 =\ \mathcal{E}[E]\ \acute{env}_0\ v_0 \\ env \sqsubseteq \acute{env}_0 \end{array} \right.$$

$$\textbf{in}\ \langle \acute{env}_0,\ r_0 \rangle$$

  From the operational semantics,
  $\langle \emptyset, F(E), \rho_0, FL_0 \rangle \xrightarrow{*} \langle \{x = E\}, e^\star, \rho_1, FL_1 \rangle$, where

$$\begin{aligned} e^\star &= exp_f[x/arg][z_1/y_1]\ldots[z_n/y_n] \\ \rho_1 &= \rho \bigcup \{\{x\},\ \{z_1\}\ldots\{z_n\}\} \\ FL_1 &= FL - \{x,\ z_1, \ldots z_n\} \end{aligned}$$

  Note that $r_f = \bot$. Since $E < F(e_0) \implies \mathcal{E}[E] \preceq E$, we have,

  - $(env_f = env_\top) \Longrightarrow \langle \emptyset, E, \rho_0, FL_0 \rangle \xrightarrow{*} \textbf{error}$
  - $(\langle \emptyset, E, \rho_0, FL_0 \rangle \xrightarrow{*} \textbf{error}) \vee$
    $(\exists\ \langle D_1, e_1, \rho_1, FL_1 \rangle \colon \langle \emptyset, E, \rho_0, FL_0 \rangle \xrightarrow{*} \langle D_1, E_1, \rho_1, FL_1 \rangle)$
    $\wedge\ (\rho_1$ *COVERS* $env_f)$

However, we have,

- $\langle \emptyset, E, \rho_0, FL_0\rangle \xrightarrow{*}$ **error**
  $\Longrightarrow \langle \{x = E\}, e^\star, \rho_1, FL_1\rangle \xrightarrow{*}$ **error**
- $(\exists\ \langle D_1, E_1, \rho_1, FL_1\rangle$: $\langle \emptyset, E_0, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_1, E_1, \rho_1, FL_1\rangle)$
  $\wedge\ (\rho_1\ \mathit{COVERS}\ env_f)$
  $\Longrightarrow \langle \{x = E\}, e^\star, \rho_1, FL_1\rangle \xrightarrow{*} \langle \{x = E_1\} \bigcup D_1, e^\star, \rho_1, FL_1\rangle$
  $\wedge\ (\rho_1\ \mathit{COVERS}\ env_f)$

Since $r_f = \perp$, $e^\star\ \mathit{COVERS}\ r_f\ \mathit{IN}\ \rho_1$ is vacuosly true.

- (Induction: $\tau_i[E] \preceq F(E) \Longrightarrow \tau_{(i+1)}[E] \preceq F(E)$)

$$\tau_{i+1}[E]\ env\ \perp = \mathbf{lcs} \left\{ \begin{array}{c} en\acute{v}_{i+1}\ v_{i+1} = \mathcal{E}[E]\ en\acute{v}_{i+1}\ v_{i+1} \\ en\acute{v}_{i+1}\ v_{i+1}\ r_{i+1} = \mathcal{E}_F[F_{i+1}]\ en\acute{v}_{i+1}\ v_{i+1}\ r_{i+1} \\ env \sqsubseteq en\acute{v}_{i+1} \end{array} \right.$$

$$\mathbf{in}\ \langle en\acute{v}_{i+1},\ r_{i+1}\rangle$$

From Lemma 13,

$$\tau_{i+1}[E]\ env\ \perp = \mathbf{lcs} \left\{ \begin{array}{c} en\acute{v}_{i+1}\ v_{i+1} = \mathcal{E}[E]\ en\acute{v}_{i+1}\ v_{i+1} \\ env \sqsubseteq en\acute{v}_{i+1} \\ \{x_0 \mapsto v_{i+1},\ y_1 \mapsto \perp, \ldots,\ y_n \mapsto \perp,\ F \mapsto \mathcal{E}_F[F_i]\} \sqsubseteq en\acute{v}_{i+1} \\ en\acute{v}_{i+1} = C[def - list]\ en\acute{v}_{i+1} \\ en\acute{v}_{i+1}\ r_{i+1} = \mathcal{E}[exp]\ en\acute{v}_{i+1}\ r_{i+1} \end{array} \right.$$

$$\mathbf{in}\ \langle en\acute{v}_{i+1},\ r_{i+1}\rangle$$

From Lemma 13,

$$\tau_{i+1}[E]\ env\ \perp = \mathbf{lcs} \left\{ \begin{array}{c} en\acute{v}_{i+1}\ v_{i+1} = \mathcal{E}[E]\ en\acute{v}_{i+1}\ v_{i+1} \\ env \sqsubseteq en\acute{v}_{i+1} \\ \{x_0 \mapsto v_{i+1},\ y_1 \mapsto \perp, \ldots,\ y_n \mapsto \perp,\ F \mapsto \mathcal{E}_F[F_i]\} \sqsubseteq en\acute{v}_{i+1} \\ en\acute{v}_{i+1}\ = C[z_1 = exp_1]\ en\acute{v}_{i+1} \\ \vdots \\ en\acute{v}_{i+1}\ = C[z_m = exp_m]\ en\acute{v}_{i+1} \\ en\acute{v}_{i+1}\ r_{i+1} = \mathcal{E}[exp]\ en\acute{v}_{i+1}\ r_{i+1} \end{array} \right.$$

$$\mathbf{in}\ \langle en\acute{v}_{i+1},\ r_{i+1}\rangle$$

Since there are no other constraints on r_{i+1}, we have,

$$\tau_{i+1}[E]\ env\ \bot = \text{lcs} \begin{cases} \acute{env}_{i+1}\ v_{i+1} = \mathcal{E}[E]\ \acute{env}_{i+1}\ v_{i+1} \\ env \sqsubseteq \acute{env}_{i+1} \\ \{x_0 \mapsto v_{i+1},\ y_1 \mapsto \bot, \ldots,\ y_n \mapsto \bot,\ F \mapsto \mathcal{E}_F[F_i]\} \sqsubseteq \acute{env}_{i+1} \\ \acute{env}_{i+1} = C[z_1 = exp_1]\ \acute{env}_{i+1} \\ \vdots \\ \acute{env}_{i+1} = C[z_m = exp_m]\ \acute{env}_{i+1} \\ \acute{env}_{i+1}\ r_{i+1} = \mathcal{E}[exp]\ \acute{env}_{i+1}\ \bot \end{cases}$$

$$\textbf{in}\ \langle \acute{env}_{i+1},\ r_{i+1} \rangle$$

From the hypothesis of the lemma, we have $(\forall e^{**} < F(e_0))\ \mathcal{E}[e^{**}] \preceq e^{**}$. From the induction hypothesis, $(\forall E < F(e_0))\ \tau_i[E] \preceq \mathcal{E}[F(E)]$. All occurrences of F in the right-hand side of the equation for $\tau_{i+1}[E]$ are bound to $\mathcal{E}_F[F_i]$. So, if $F(e)$ occurs in any of exp, $exp_1, \ldots exp_m$, we have $\tau_i[e] \preceq F(e)$. Using Lemmas 9, 10, 11, 12

$$\begin{aligned} \mathcal{E}[exp_1] &\preceq exp_1 \\ &\vdots \\ \mathcal{E}[exp_m] &\preceq exp_m \\ \mathcal{E}[exp] &\preceq exp \end{aligned}$$

when all occurrences of F in exp, $exp_1, \ldots exp_m$ are bound to $\mathcal{E}_F[F_i]$. Proof now is similar to the Lemma 10 , except that the operational rule $\langle D_0, e, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_1, e_1, \rho_1, FL_1\rangle \Longrightarrow \langle D_0 \bigcup \{x = e\}, e^{\star}, \rho_0, FL_0\rangle$ $\langle D_0 \bigcup \{x = e_1\}, e^{\star}, \rho_1, FL_1\rangle$ is used after the reduction sequences corresponding to $\mathcal{E}[exp_1] \ldots \mathcal{E}[exp_m]$.

These lemmas complete the cases of the structural induction nneeded for the adequacy proof.

Theorem 4. $(\forall e:\ \mathcal{E}[e] \preceq e)$.

Proof: (By structural induction).

- (Base cases)

  1. $e = c$. Let ρ_0 *COVER env*.
     $\mathcal{E}[c]\ env\ \bot\ =\ \langle env,\ c\rangle$. Since no basic constant denotes $\top$, it suffices to note that $\langle D_0, c, \rho_0, FL_0\rangle$ satisfies required properties.
  2. $e = y$. Let ρ_0 *COVER env*.

     - ($env[y] = \bot$). $\langle D_0, y, \rho_0, FL_0\rangle$ satisfies required properties.
     - ($env[y] = b$). $\langle D_0, y, \rho_0, FL_0\rangle \xrightarrow{*} \langle D_0, b, \rho_0, FL_0\rangle$ which satisfies required properties.
     - ($env[y] = A$). Follows from the fact that ρ_0 *COVERS env*.

- (Induction). Use Lemmas 9, 10, 11, 12, 15.

6.3 Proof of full-abstraction

In full-abstraction we aim to establish that the denotational semantics is an accurate guide to program behaviour *in all contexts.* Since the interpreter works with operational configurations, the contexts available to the interpreter are definition and expression contexts. Let $D[]$ denote a definition context with one hole. Let $C[]$ denote an expression context with one hole. We define an operational preorder that expresses the relative contextual behaviour of syntactic expressions as follows.

Definition 13. $e_1 \sqsubseteq_{op} e_2$ if for all definition contexts $D[]$ and for all expression contexts $C[]$,

- $\langle D[e_1], C[e_1], \Phi, FL\rangle \rightarrow b$, where b is a integer implies
  $\langle D[e_2], C[e_2], \Phi, FL\rangle \rightarrow b$ or $\langle D[e_2], C[e_2], \Phi, FL\rangle \rightarrow error$.

- $\langle D[e_1], C[e_1], \Phi, FL\rangle \rightarrow error$ implies
  $\langle D[e_2], C[e_2], \Phi, FL\rangle \rightarrow error$

The basic results of this section are that the approximation relation between the meanings of terms in the domain accurately reflects the operational preorder. The first theorem below states that the denotational order implies the operational preorder. This is essentially a consequence of the fact that one-step reduction preserves meaning.

Theorem 5. The denotational semantics is inequationally adequate i.e

$$\mathcal{E}[\![e_1]\!] \sqsubseteq \mathcal{E}[\![e_2]\!] \Longrightarrow e_1 \sqsubseteq_{op} e_2$$

Let $\mathcal{E}[\![e_1]\!] \sqsubseteq \mathcal{E}[\![e_2]\!]$ and $\langle D[e_1], C[e_1], \Phi, FL\rangle \rightarrow b$. Consider

$$\mathcal{M}[\![\langle D[e_1], C[e_1, \Phi, FL\rangle]\!] \quad \perp \ \perp.$$

Since one step reduction preserves meaning,
$\mathcal{M}[\![\langle D[e_1], C[e_1, \Phi, FL\rangle]\!] \quad \perp \ \perp = env, b$ for some env.
Since the context operations are monotone, we have
$env, b \sqsubseteq \mathcal{M}[\![\langle D[e_2], C[e_2], \Phi, FL\rangle]\!] \quad \perp \ \perp.$

Let $x_1 = E_1[], \ldots \ x_n = E_n[]$ be the equations in the definition context $D[]$. Define a function $F(x_1, \ldots, x_n)$ as follows:

$$F(x_1, \ldots x_n) \ = \ \begin{cases} x_1 = E_1[e_2] \\ \quad \ldots \\ x_n = E_n[e_2] \end{cases} \\ \textbf{in } C[e_2]$$

Note that $\langle \Phi, F(x_1, \ldots x_n), \Phi, FL\rangle \rightarrow \langle D[e_2], C[e_2], \Phi, FL\rangle$. Hence, we have
$\mathcal{E}[\![F(x_1, \ldots x_n)]\!] \quad \perp \ \perp = \mathcal{M}[\![\langle D[e_2], C[e_2, \Phi, FL\rangle]\!] \quad \perp \ \perp$
because one-step reduction preserves meaning. Hence,
$\perp, b \sqsubseteq \mathcal{E}[\![F(x_1, \ldots x_n)]\!] \quad \perp \ \perp$
Hence, we have, either

- $\langle \Phi, F(x_1, \ldots x_n), \Phi, FL\rangle \rightarrow b$ or
- $\langle \Phi, F(x_1, \ldots x_n), \Phi, FL\rangle \rightarrow error$

From the Church-Rosser property of the operational semantics, we have one of

- $\langle D[e_2], C[e_2], \Phi, FL\rangle \rightarrow b$ or
- $\langle D[e_2], C[e_2], \Phi, FL\rangle \rightarrow error$

The equivalence of the two orders is full-abstraction. It is essentially a consequence of the fact that all the finite elements of the domain are expressible as the meanings of expressions, as in Plotkin's proof of full-abstraction for PCF [7]. The crux of the proof below is the construction of contexts that can semantically distinguish two different expressions.

Theorem 6. The denotational semantics is fully-abstract i.e

$$\mathcal{E}[\![e_1]\!] \sqsubseteq \mathcal{E}[\![e_2]\!] \Longleftrightarrow e_1 \sqsubseteq_{op} e_2$$

Proof
The forward implication was proved in the previous theorem. For the reverse impication consider the case when $\mathcal{E}[\![e_1]\!] \not\sqsubseteq \mathcal{E}[\![e_2]\!]$. Since the semantic domains are algebraic
$\mathcal{E}[\![e_1]\!] \not\sqsubseteq \mathcal{E}[\![e_2]\!] \Longrightarrow$
$(\exists \langle env_1, v_1 \rangle,\ \langle env_2, v_2 \rangle)$
$[f_{\langle env_1, v_1 \rangle \Rightarrow \langle env_2, v_2 \rangle} \sqsubseteq \mathcal{E}[\![e_1]\!] \ \wedge\ f_{\langle env_1, v_2 \rangle \Rightarrow \langle env_2, v_2 \rangle} \not\sqsubseteq \mathcal{E}[\![e_2]\!]]$
where $f_{\langle env_1, v_1 \rangle \Rightarrow \langle env_2, v_2 \rangle}$ is the step function in $V \times ENV \Rightarrow V \times ENV$ defined as

$$f_{\langle env_1, v_2 \rangle \Rightarrow \langle env_1, v_2 \rangle} env\ v \ = \ \begin{cases} env, v & \textbf{if } env_1, v_1 \not\sqsubseteq env, v \\ env \bigsqcup env_2, v \bigsqcup v_2 & \textbf{otherwise} \end{cases}$$

Since env_1 is finite, it can be represented by a finite set of equations, say E. Similarly, since v_1 is a finite value the semantic equation $x = v_1$ can be coded as a finite set of syntactic equations that set x to v_1. Let this set of equations be named E'. For the same reasons, there is an operational expression that corresponds to $v_2 \sqsubseteq x \wedge env_2 \sqsubseteq env$, say $C[x]$.

In the light of the previous remarks the following function definition is a valid expression in the syntax of Id-Noveau.

$$F(x) \ = \ \begin{cases} E \\ E' \\ \end{cases}$$
$$\textbf{in } if\ C[x]\ then\ 0\ else\ 1.$$

We shall prove that $\langle \Phi, F(\cdot), \Phi, FL\rangle$ is the required operational context to distinguish e_1 and e_2. The proof proceeds in two stages:

- First, we deduce that $\langle \Phi, F(e_1), \Phi, FL\rangle$ reduces to 0 or to error. $\mathcal{E}[\![F(e_1)]\!]\ \perp\ \perp = \mathcal{M}[\![\langle \Phi, F_{e_1}, \Phi, FL\rangle]\!]\ \perp\ \perp =$ $\mathcal{M}[\![\langle G, if\ E'\ then\ 0\ else\ 1, \phi, FL\rangle]\!]\ \perp\ \perp$, where $G = E \bigcup E' \bigcup \{x = e_1\}$. However, we have

$$\mathcal{M}[\![\langle G, if\ E'\ then\ 0\ else\ 1, \phi, FL\rangle]\!]\ \perp\ \perp \quad = \quad \text{lcs}\begin{cases} \mathcal{E}[\![E]\!]env = env & 1 \\ \mathcal{C}[\![x = e_1]\!]\ env = env & 2 \\ \mathcal{E}[\![E']\!]env = env & 3 \\ \mathcal{E}[\![C[x]]\!]\ env\ \ a = a & 4 \end{cases} \ \textbf{in}\ \langle env, a\rangle$$

  Equation 1 merely asserts that $env_1 \sqsubseteq env$. Equation 3 ensures that $v_1 \sqsubseteq env[x]$. Now equation 2 ensures that $env_2 \sqsubseteq env$ and $env[x] \sqsubseteq v_2$. So, we deduce that $env_2, 0 \sqsubseteq \mathcal{E}[\![F(e_1)]\!]\ \perp\ \perp$. Also, $\langle env_2, 0\rangle$ is a finite element in $V \times ENV$. So, we deduce that the result part of $\mathcal{E}[\![F(e_1)]\!]\ \perp\ \perp$ is 0 or T. Hence, $\langle \Phi, F(e_1), \Phi, FL\rangle$ reduces to 0 or to error.

- Let $\langle \Phi, F(e_2), \Phi, FL\rangle$ reduce to *error*. This means that $\mathcal{E}[\![F(e_1)]\!]\ \perp\ \perp =$ T. That implies that the least common solution of equations 1, 2 and 3 is T. Hence, we deduce that $\mathcal{E}[\![e_2]\!]\ env_1\ v_1 = T$, which contradicts the fact that $f_{\langle env_1, v_1\rangle \Rightarrow \langle env_2, v_2\rangle} \not\sqsubseteq \mathcal{E}[\![e_2]\!]$. So, $\langle \Phi, F(e_2), \Phi, FL\rangle$ does not reduce to *error*.

- Let $\langle \Phi, F(e_2), \Phi, FL\rangle$ reduce to 0. This means that $\perp, 0 \sqsubseteq \mathcal{E}[\![F(e_1)]\!]\ \perp\ \perp$. That implies that the least common solution of equations 1, 2 and 3 is greater than $\langle env_2, v_2\rangle$. Hence, we deduce that $env_2, v_2 \sqsubseteq \mathcal{E}[\![e_2]\!]\ env_1\ v_1$, which contradicts the fact that $f_{\langle env_1, v_1\rangle \Rightarrow \langle env_2, v_2\rangle} \not\sqsubseteq \mathcal{E}[\![e_2]\!]$. Hence, $\langle \Phi, F(e_2), \Phi, FL\rangle$ does not reduce to 0.

This completes the proof of full abstraction for Cid.

7 Conclusions

There are three main results in this paper. First, we gave an operational semantics for such a language using Plotkin-style structural operational semantics. Second, we gave an abstract, denotational semantics using closure operators. The denotational semantics is couched in terms of solving equations and is rather different from the operational semantics. Finally, we showed that the denotational semantics is fully abstract with respect to the operational semantics, thus showing that the abstract semantics fits precisely with the concrete operational semantics. We feel that this is the first satisfactory *abstract* semantic account of what it means to add logic variables to functional languages. Lindstrom's account of logic variables, while couched in denotational formalism, encoded operational notions, such as the propagation of demand tokens, in the denotational semantics [2].

There are a number of ways in which these results can be extended. It may be possible to extend the denotational semantics to a higher-order language. The full abstraction result will, however, much harder to establish for such a language. Another problem we have not considered is modeling termination. Termination is important operationally because a program may produce the value 2, for example, at its output and then 'refine' it to the error value later in the execution, if some unification operation fails. Thus, until it is known that the program has terminated, we cannot know whether an output is final or is subject to further refinement as computation proceeds. It would be interesting to extend our semantics to capture this notion of termination. Finally, there remains the problem of reasoning about functional languages with logic variables. The soundness of the proof rules in such a system can be verified using our denotational semantics.

References

[1] 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.

[2] G. Lindstrom. Functional programming and the logic variable. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming languages*, 1985.

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

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

[5] K. Mulmuley. *Full Abstraction and Semantic Equivalence.* ACM Distinguished Dissertation Series. MIT Press, 1987. CMU Ph.D. dissertation.

[6] R. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical Report CSG Memo 265, MIT Laboratory for Computer Science, 1986.

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

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

[9] Dana Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, 1976.

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

[11] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4), 1982.

[12] Allen Stoughton. *Fully Abstract Models of Programming Languages.* PhD thesis, University of Edinburgh, 1986. Available as CST-40-86.

[13] J. E. Stoy. *Denotational Semantics.* MIT Press, 1977.