

# A Fully Abstract Semantics for a First-order Functional Language with Logic Variables

Radha Jagadeesan  
Keshav Pingali  
Computer Science Department  
Cornell University

Prakash Panangaden  
Computer Science Department  
McGill University.

August 28, 1991

## Abstract

There is much interest in combining the functional and logic programming paradigms; in particular, there have been several proposals for adding logic variables to functional languages, since that permits incremental construction of data structures through constraint intersection. While it is straight-forward to give an abstract semantics for functional languages and for logic languages, it has proven surprisingly difficult to give a proper semantic account of functional languages with logic variables. In this paper, we present a first-order functional language with logic variables and give its meaning using a structural operational semantics. We also give it a denotational semantics, using a novel technique involving closure operators on a Scott domain. Finally, we show that these two semantics correspond in the strongest possible way — we show that the denotational semantics is fully abstract with respect to the operational semantics. The techniques developed in this paper are quite general, and can be used to give semantics to any constraint-based logic programming languages. Our results can also be interpreted as a generalization of Kahn semantics for dataflow networks in which processes not only exchange messages, but have access to a shared global address space in which variables are bound through constraint intersection.

Categories and Subject Descriptors: D.1.1 [Programming Techniques]: Functional Programming; D.3.1 [Programming Languages]: Formal Definitions and Theory - semantics; D.3.2 [Programming Languages]: Dataflow Languages; F.3.2 [Theory of Computation]: Semantics of Programming Languages - denotational semantics; F.4.1 [Theory of Computation]: Mathematical Logic - logic programming

General Terms: Design, Languages, Theory.

Additional Key Words and Phrases: Declarative languages, functional languages, logic variables, semantics, full abstraction.

## 1 Introduction

Functional and logic language programs can be given semantics as functions and relations over values such as integers and booleans, without reference to a global store updated sequentially by program statements. These functions and relations can be computed in parallel with the guarantee that the output of the program does not depend on the order in which computations are performed. For this reason, there is much interest in using functional and logic languages to program parallel machines [?, ?, ?, ?, ?, ?, ?, ?].

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 representation of integers as terms built from the functors zero and successor, for instance, is not recommended for programmers who worry about how fast their programs run. Every ‘real’ logic programming language includes facilities for evaluating arithmetic and logical functions, as well as an operator (such as *is* or  $:=$ ) for binding an identifier to the output of the function [?]. 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 computational features 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). The efficient implementation of full-blown OR-parallelism is difficult, and even within the logic programming community, there are alternatives, such as committed choice non-determinism, that are being actively investigated [?, ?, ?]. On the other hand, logic variables can be introduced quite easily into a functional language and the merits of doing so have been remarked on by Lindstrom and others; in particular, logic variables permit elegant coding of constraint-based algorithms such as Milner’s polymorphic type deduction algorithm and 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 which is produced as the result of evaluating a single applicative expression, just like an integer or floating-point number. This works well when the data structure can be built in a bottom-up manner: first,

---

<sup>1</sup>Correspondence regarding this paper should be addressed to Keshav Pingali. Keshav Pingali was supported by an NSF Presidential Young Investigator’s award, NSF grant CCR-9008526 and by grants from the Math Sciences Institute, Cornell, the Digital Equipment Corporation and the Hewlett-Packard Corporation. Radha Jagadeesan and Prakash Panangaden were supported by NSF grant CCR-8818979;

the components of the data structure are constructed, and then these components are assembled together to produce the desired data structure. However, this does not work for flat, random-access data structures such as arrays and matrices. The construction of such data structures in a functional language can be obscure and inefficient since putting together a matrix of size  $n \times n$  may involve making  $n^2$  intermediate copies of the matrix [?]. Logic variables provide an elegant solution to this problem because they allow the programmer to define an array incrementally without requiring these intermediate copies: the programmer can allocate a matrix of the desired size, in which each element of the matrix contains an uninitialized logic variable, and bind each of these logic variables incrementally in the program. For example, a matrix can be constructed by two procedures, one of which instantiates variables on the boundary while the other instantiates variables in the interior. In this way, arrays can be constructed incrementally without the copy overhead of purely functional arrays. These observations motivated the design of *Id Nouveau* which is a functional language with logic variables[?, ?]. *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 [?].

To understand the subtleties of combining the functional and logic paradigms, we felt it was important to define abstract semantics for *Id Nouveau* programs that did not involve operational notions such as token pushing that were present in other abstract semantics for this class of languages [?]. It is well-known that standard denotational techniques can be used to give semantics to pure functional and logic languages in terms of functions and relations on domains [?, ?]. To our surprise, such an abstract semantic account of a functional language with logic variables turned out to be much more difficult. The operational semantics of such a language involves both reduction, as in functional languages, and constraint solving through unification, as in logic languages. Unification of logic variables makes them aliases for the same object. The traditional denotational technique for handling aliasing is to use a two-level store [?], but this technique results in loss of abstraction since the manipulations of the store are exposed in the semantics. Another complication is that these languages are inherently parallel in the sense that any correct interpreter for this class of languages must either be parallel or must simulate parallelism, as we show in Section ?? . Instantiation of a logic variable has the flavor of a globally-visible side-effect, and modeling global side-effects in the presence of concurrency is difficult, usually requiring the use of techniques like powerdomains [?].

The contribution of this paper is the insight that these problems can be overcome by basing the semantics on equation solving and relating equation solving to the notion of closure operators on Scott domains [?]. More precisely, we give an operational and a denotational semantics for the first-order subset of *Id Nouveau*, and show that the denotational semantics is fully abstract with respect to the operational semantics, which is a surprisingly strong result since the operational and denotational semantics have little similarity to each other. The rest of this paper is organized as follows. In Section 2, we introduce the first-order subset of

Id Nouveau through the use of two programming examples which illustrate the subtle interaction between concurrency and logic variable instantiation. 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 and constraint-solving through unification. In Section 5, we present an abstract (denotational) semantics for Cid which abstracts away from operational details such as unification and parallel evaluation and which is couched in terms of equation solving. We show that equation solving is tied intimately to the notion of *closure operators* on Scott domains [?]. In Section 6, we show that the denotational semantics is fully abstract with respect to the operational semantics [?, ?]. Details of this proof are presented in the appendix.

The results in this paper should be of interest to the functional and logic programming community as well as to the semantics community. The functional languages community has been skeptical about logic variables because of the fear, among other things, that their incorporation would result in the loss of clean abstract semantics; for these researchers, our results show that that fear is unfounded. For the logic programming community, and particularly the constraint logic programming community, we offer new techniques for giving semantics to languages such as Guarded Horn Clauses [?], Qute [?] and concurrent constraint programming languages in general. Up till now, these languages have been defined by specifying an informal operational semantics. In particular, a semantic account of deterministic constraint programming languages is straight-forward since they incorporate only AND-parallelism, and our techniques can be applied directly. The techniques we develop in this paper have been adopted by Saraswat et al to give abstract semantics to constraint languages [?] and by Mantha, Lindstrom and George to give semantics to a lazy functional language with logic variables [?]. For the semantics community, we show a full abstraction result for a parallel programming language. Most full abstraction results in the literature are routine because the denotational semantics is an encoding of the operational semantics. In contrast, the operational and denotational semantics for our language are completely different, and proving a full abstraction result is non-trivial. For researchers interested in the semantics of concurrency, we point out that the semantics in this paper extends the equation-solving paradigm that underlies Kahn semantics for dataflow networks [?] to a more expressive setting in which processes manipulate shared memory locations. Kahn's dataflow networks communicate by sending messages on channels, and message transmission is monotonic in the sense that a message cannot be deleted or recalled once it has been sent. Our framework extends this model with monotonic shared memory in which globally visible names are bound to values through unification.

## 2 Informal Introduction to the Language

This section introduces Id Nouveau and its operational semantics informally through a number of programming examples. The operational semantics we present here is a simplified version of the formal operational semantics in Section 4. For a complete description of Id Nouveau, we refer the reader to [?]. We assume that the reader is familiar with functional languages; therefore, we will begin by describing the array construct that uses logic variables. 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  $A = \text{array}(5)$  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 length  $n$  containing a permutation of the integers  $1..n$ , build a new array  $A$  of length  $n$  such that  $A[B[i]] = i$ . This is called an inverse permutation because the result array  $A$  contains a permutation of the integers  $1..n$ , and when the operation is repeated with  $A$  as an argument, the original permutation is returned. It is straight-forward to write a program for this problem in our language.

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

{A = [L1,L2,L3];

```

```

L2 = 1;
L1 = 2;          -----(4)
L3 = 3;
in A}

```

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

Unlike in functional languages, the array *A* has not been produced as the result of evaluation of a single expression - instead, it has been defined incrementally by the co-operation of a number of definitions in the program. Abstractly, this process can be viewed as *constraint intersection*. Consider, for example, expression (3). Each of the definitions in this expression can be thought of as constraints on the array *A*. The first definition is a constraint that asserts that *A* is an array of length 3. The second definition is a constraint that asserts that the second element of *A* is 1. In this way, each definition can be interpreted as a constraint on the value of *A* and the resulting value of *A* is obtained by intersecting all these constraints together. 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 as though it was 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,4);
in A}

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

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

```

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

```

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 |
    if exp1 then exp2 else exp3 |
    array(exp) | exp1[exp2] | Fi(exp1)

```

Figure 1: Syntax of Cid

### 3 Cid: a subset of Id Nouveau

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

Figure 1 describes the syntax of Cid. The main differences between Id Nouveau, as presented informally in Section ??, and the 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 identifier. This is reasonable if we think of definitions as constraints. In our development in later sections, we will sometimes use a prefix `cond` operator in place of the infix `if-then-else` in the syntax, to avoid confusion with the semantic conditional function.

To meet these syntactic restrictions, the `fill-even` procedure discussed in Section ?? can be re-written

as follows:

```
def new-fill-even(X,h,i) = {t = X[2*i];
                           t = X[2*i-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 convention used in logic programming: the body of a procedure is a single scope and the formal parameters of the procedure are in the same scope. Finally, to reduce the need for ellipses and subscripts, we will require that a procedure have exactly one formal parameter. 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. If `F` is a function,  $\text{arg}_F$ ,  $\text{locals}_F$ ,  $\text{defs}_F$  and  $\text{exp}_F$  denote the formal parameter, local variables, 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 quadruple  $\langle D, e, \rho, FL \rangle$  where  $D$  is a set of definitions,  $e$  is an expression,  $\rho$  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  $\rho$  is a (possibly empty) set of *alias-sets* where an alias-set is an equivalence class of base values. For example,  $\{x, y, z\}$ ,  $\{x, y, 4\}$  and  $\{x, y, [L1, L2]\}$  are alias-sets. If  $b1$  and  $b2$  are two base values in the same alias-set, then occurrences of  $b1$  in  $D$  and  $e$  may be replaced by  $b2$  without changing the meaning of the program.

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.

$x, L \in \text{Id}$ = set of identifiers	$c \in \text{Constant}$ = set of constants
$A \in \text{Array} ::= [x_1, \dots, x_n]$	$B \in \text{Base-value} ::= x c Ar$
$A \in \text{Alias-set} ::= \{B_1, \dots, B_n\}$	$\rho \in \text{Environment} ::= \phi \{A_1, \dots, A_n\}$
$e \in \text{expression}$	$D \in \text{Defs} ::= \phi \{\text{def}_1, \dots, \text{def}_n\}$ (defined by the syntax of the language)
$FL \in \text{Free-list} = \mathcal{P}(\text{Id})$	$C \in \text{Configurations} ::= \langle D, e, \rho, FL \rangle   \text{Error}$

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

## 4.2 Unification

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

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

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

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

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

**Definition 2**  $\rightsquigarrow$  is a binary relation on environments defined as follows:

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

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

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

**Theorem 1** *The relation  $\rightsquigarrow^*$  has the following properties [?]:*

1. *If  $\rho_1 \rightsquigarrow^* \rho_2$  and  $\rho_1 \rightsquigarrow^* \rho_3$  then  $\rho_2 \rightsquigarrow^* \rho_4$  and  $\rho_3 \rightsquigarrow^* \rho_4$  for some  $\rho_4$ .*
2. *There is no infinite sequence of distinct environments  $\rho_i$  such that  $\rho_i \rightsquigarrow \rho_{i+1}$  for all  $i$ .*
3. *For any environment  $\rho$ , there is a unique, irreducible  $\rho_1$  such that  $\rho \rightsquigarrow^* \rho_1$ .*

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

Recall that a configuration is a quadruple  $\langle D, e, \rho, FL \rangle$ . 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  $\rho$  and this environment is reduced completely. The resulting environment incorporates all the constraints in  $\rho$  and in the definition.

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

### 4.3 Rewrite rules

The rewrite rules for configurations are specified in terms of a binary relation  $\rightarrow$  on the set of configurations. In any program  $P$ , let  $exp_P$  be the expression to be evaluated. The initial configuration for program  $P$  is  $\langle \phi, exp_P, \phi, Id \rangle$ . 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  $\rho$  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 is an element of exactly one alias-set. This leads to the following definition.

**Definition 4** *If  $\langle D, e, \rho, FL \rangle$  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  $\rho(x)$  is defined by cases on  $A$ :*

1. *All the elements of  $A$  are identifiers. In that case,  $\rho(x)$  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  $\rho(x)$  to be  $c$ .*
3. *At least one element of  $A$  is an array. Since  $A$  is consistent, the elements of  $A$  are either identifiers or arrays of the same length.  $\rho(x)$  could be defined to be any one of these arrays. To be precise, place a lexicographical ordering on identifiers and let  $\rho(x)$  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  $\rho$  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  $F(\mathbf{array}(2))$ . If  $\mathbf{array}(2)$  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  $\mathbf{array}(2)$  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  $\mathbf{exp}_F$  and adding the definitions in  $\mathbf{defs}_F$  to the definitions in  $D$ . The formal parameter  $x$  and local variables  $y_i$  of the function are renamed to new identifiers to avoid name clashes. Since

the actual parameter  $e$  need not be a base value, a definition of the form  $x = e$  is added to the definitions in the configuration.

#### 4.4 Properties of the Rewrite Rules

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

**Definition 1**  $\langle D_1, e_1, \rho_1, FL_1 \rangle$  and  $\langle D_2, e_2, \rho_2, FL_2 \rangle$  are alpha-equivalent if  $\exists x_1 \dots x_n \in FL_1, ; y_1 \dots y_n \in FL_2$  such that  $FL_1 - \{x_1 \dots x_n\} = FL_2 - \{y_1 \dots y_n\}$ , and replacing  $x_1 \dots x_n$  by  $y_1 \dots y_n$  in  $D_1, e_1, \rho_1$  gives  $D_2, e_2, \rho_2$  respectively.

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

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

**Lemma 1** If  $\langle D_0, e_0, \rho_0, FL_0 \rangle \longrightarrow conf_1$  and  $\langle D_0, e_0, \rho_0, FL_0 \rangle \longrightarrow conf_2$ , then

1. If  $conf_1 = \text{error}$ ,  $conf_2 \longrightarrow \text{error}$
2. If  $conf_2 = \text{error}$ ,  $conf_1 \longrightarrow \text{error}$
3. Let  $conf_1 = \langle D_1, e_1, \rho_1, FL_1 \rangle$ ,  $conf_2 = \langle D_2, e_2, \rho_2, FL_2 \rangle$ , and  $(FL_0 - FL_1) \cap (FL_0 - FL_2) = \Phi$ . Then one of the following holds:
  - (a)  $conf_1 \xrightarrow{\alpha} conf_2$
  - (b)  $conf_1 \longrightarrow \text{error}$ ,  $conf_2 \longrightarrow \text{error}$
  - (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 follows immediately from a case-by-case analysis of the rules in Figure ???, and is omitted.

□

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

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

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

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

## 4.5 Interpreter for Cid programs

Given the rules in Figure ?? and the results in Section ??, it is straight-forward to design an interpreter for Cid programs. The interpreter rewrites configurations, selecting any enabled transition at each step. To guarantee progress, there must be some notion of fair-scheduling in the sense that no enabled transition is postponed indefinitely. The detection of enabled transitions is straight-forward. Notice that a function application that is not within a conditional can be performed immediately, even if the actual parameter has not been reduced to a base value. For all other constructs, one or more arguments must become base values, and this condition can be checked by the interpreter whenever it rewrites a sub-expression to a base value. The dataflow implementation uses this strategy.

## 5 Denotational Semantics

Although the rewrite rules in Figure ?? are straight-forward, the operational model of Cid is more complicated than that of conventional languages like FORTRAN or PASCAL. These complications arise from the aliasing of names introduced by unification and from the need to interleave computations in different parts of the program - in effect, logic variable instantiation is like a globally visible side-effect in an operational model with concurrency. It is well-known that aliasing, globally visible side-effects and concurrency introduce a great deal of complexity into denotational models of languages - aliasing is modeled denotationally by introducing a store and separating the mapping of names to locations from the mapping of locations to values, while side-effects and concurrency are modeled using power-domains [?, ?]. Therefore, the main result of this section is rather unexpected - we show that Cid can be given a simple, abstract semantics couched purely in terms of equations and equation solving. In fact, we show in Section ?? that this semantics is fully abstract with respect to the operational semantics, which is the strongest ‘correspondence result’ between the two semantics that we could hope to show.

Expressions:	
Identifiers:	1. $\langle D, x, \rho, FL \rangle \rightarrow \langle D, \rho(x), \rho, FL \rangle$ (if $\rho(x)$ is defined)
Basic Operations:	1. $\frac{\langle D, e_1, \rho, FL \rangle \rightarrow \langle D^*, e_1^*, \rho^*, FL^* \rangle}{\langle D, e_1 \text{ op } e_2, \rho, FL \rangle \rightarrow \langle D^*, e_1^* \text{ op } e_2, \rho^*, FL^* \rangle}$ 2. $\frac{\langle D, e_2, \rho, FL \rangle \rightarrow \langle D^*, e_2^*, \rho^*, FL^* \rangle}{\langle D, e_1 \text{ op } e_2, \rho, FL \rangle \rightarrow \langle D^*, e_1 \text{ op } e_2^*, \rho^*, FL^* \rangle}$ 3. $\langle D, m \text{ op } n, \rho, FL \rangle \rightarrow \langle D, r, \rho, FL \rangle$ (where $r = m \text{ op } n$ )
Conditional:	1. $\frac{\langle D, e_1, \rho, FL \rangle \rightarrow \langle D^*, e_1^*, \rho^*, FL^* \rangle}{\langle D, \text{cond}(e_1, e_2, e_3), \rho, FL \rangle \rightarrow \langle D^*, \text{cond}(e_1^*, e_2, e_3), \rho^*, FL^* \rangle}$ 2. $\langle D, \text{cond}(\text{true}, e_2, e_3), \rho, FL \rangle \rightarrow \langle D, e_2, \rho, FL \rangle$ 3. $\langle D, \text{cond}(\text{false}, e_2, e_3), \rho, FL \rangle \rightarrow \langle D, e_3, \rho, FL \rangle$
Array:	1. $\frac{\langle D, e, \rho, FL \rangle \rightarrow \langle D^*, e^*, \rho^*, FL^* \rangle}{\langle D, \text{array}(e), \rho, FL \rangle \rightarrow \langle D^*, \text{array}(e^*), \rho^*, FL^* \rangle}$ 2. $\langle D, \text{array}(n), \rho, FL \rangle \rightarrow \langle D, [L1, \dots, Ln], \rho^*, FL^* \rangle$ (where $L1, \dots, Ln \in FL$ , $\rho^* = \rho \cup \{\{L1\}, \dots, \{Ln\}\}$ , and $FL^* = FL - \{L1, \dots, Ln\}$ )
Array Selection:	1. $\frac{\langle D, e_1, \rho, FL \rangle \rightarrow \langle D^*, e_1^*, \rho^*, FL^* \rangle}{\langle D, e_1[e_2], \rho, FL \rangle \rightarrow \langle D^*, e_1^*[e_2], \rho^*, FL^* \rangle}$ 2. $\frac{\langle D, e_2, \rho, FL \rangle \rightarrow \langle D^*, e_2^*, \rho^*, FL^* \rangle}{\langle D, e_1[e_2], \rho, FL \rangle \rightarrow \langle D^*, e_1[e_2^*], \rho^*, FL^* \rangle}$ 3. $\langle D, [L1, \dots, Ln][i], \rho, FL \rangle \rightarrow \langle D, Li, \rho, FL \rangle$ (where $1 \leq i \leq n$ )
Application:	1. $\langle D, F(e), \rho, FL \rangle \rightarrow \langle D^*, e_1, \rho^*, FL^* \rangle$ (where $D^* = D \cup \{x = e\} \cup \text{defs}_F[x/\text{arg}_F][y_i/\text{locals}_F]$ ( $x, y_i \in FL$ ) $e_1 = \text{exp}_F[x/\text{arg}][y_i/\text{locals}_F]$ , $\rho^* = \rho \cup \{\{x\}, \{y_i\}\}$ and $FL^* = FL - \{x, y_i\}$ )
Definitions:	1. $\frac{\langle D, e, \rho, FL \rangle \rightarrow \langle D^*, e^*, \rho^*, FL^* \rangle}{\langle D \cup \{x = e\}, e_1, \rho, FL \rangle \rightarrow \langle D^* \cup \{x = e^*\}, e_1, \rho^*, FL^* \rangle}$ 2. $\langle D \cup \{x = y\}, e, \rho, FL \rangle \rightarrow \langle D, e, \mathcal{U}(\rho, \{x, y\}), FL \rangle$ (if $\mathcal{U}(\rho, \{x, y\})$ is consistent) $\langle D \cup \{x = y\}, e, \rho, FL \rangle \rightarrow \text{Error}$ (otherwise) 3. $\langle D \cup \{x = c\}, e, \rho, FL \rangle \rightarrow \langle D, e, \mathcal{U}(\rho, \{x, c\}), FL \rangle$ (if $\mathcal{U}(\rho, \{x, c\})$ is consistent) $\langle D \cup \{x = c\}, e, \rho, FL \rangle \rightarrow \text{Error}$ (otherwise) 4. $\langle D \cup \{x = [L1, \dots, Ln]\}, e, \rho, FL \rangle \rightarrow \langle D, e, \mathcal{U}(\rho, \{x, [L1, \dots, Ln]\}), FL \rangle$ (if $\mathcal{U}(\rho, \{x, [L1, \dots, Ln]\})$ is consistent) $\langle D \cup \{x = [L1, \dots, Ln]\}, e, \rho, FL \rangle \rightarrow \text{Error}$ (otherwise)

Figure 2: Operational Semantics of Cid

## 5.1 Informal Introduction to the Abstract Semantics

To understand the intuition behind the denotational semantics, let us consider once again the small Cid program discussed in Section ??:

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

If  $B$  is the domain of basic values such as integers and booleans, consider the domain of both basic values and arrays, which can be described informally by the following domain equation:

$$W = B + W + W \times W + W \times W \times W + \dots$$

In the infinite sum, the component  $B$  represents basic values, the component  $W$  represents arrays of length 1, the component  $W \times W$  represents arrays of length 2 etc. Notice that array elements come from the domain  $W$  itself - therefore, array elements can be arrays themselves, and the domain includes ‘infinitely nested’ arrays such as the one discussed in Section ?. To this domain, we add an element labeled  $\top$  which is a special value that models error, the result of (contradictory) definitions such as

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

A pictorial representation of the resulting domain, which we call  $V$ , is shown in Figure ?. Values in  $V$  are ordered the usual way. Arrays of different lengths are incomparable and if  $a_1$  and  $a_2$  are two arrays of the same length, we say that  $a_1 \sqsubseteq a_2$  if  $a_2$  can be obtained by replacing occurrences of  $\perp$  in  $a_1$  by other values from  $W$ . For example, the least defined array of length 3 is  $[\perp, \perp, \perp]$  and it is below  $[2, \perp, \perp]$  which in turn is below  $[2, \perp, 3]$  etc. The error element  $\top$  is above all other values in  $V$ . The array constructor is strict in  $\top$  in the sense that any array with  $\top$  as an element is identified with  $\top$ .

As we discussed in Section ?, Cid programs can be viewed abstractly in terms of constraints. A definition of the form  $x = \text{array}(3)$  can be viewed as a constraint that gives partial information about the array  $x$  - any array of length 3, and the value  $\top$ , which satisfies all constraints trivially, satisfy this constraint. Similarly, the definition  $x[1] = 2$  is satisfied by any value from  $V$  that represents an array in which the first element is 2, and by  $\top$ . Thus, a definition can be treated as a constraint which has a set of possible solutions, and composition of definitions can be treated as simultaneous constraint imposition which corresponds to intersection of these sets.

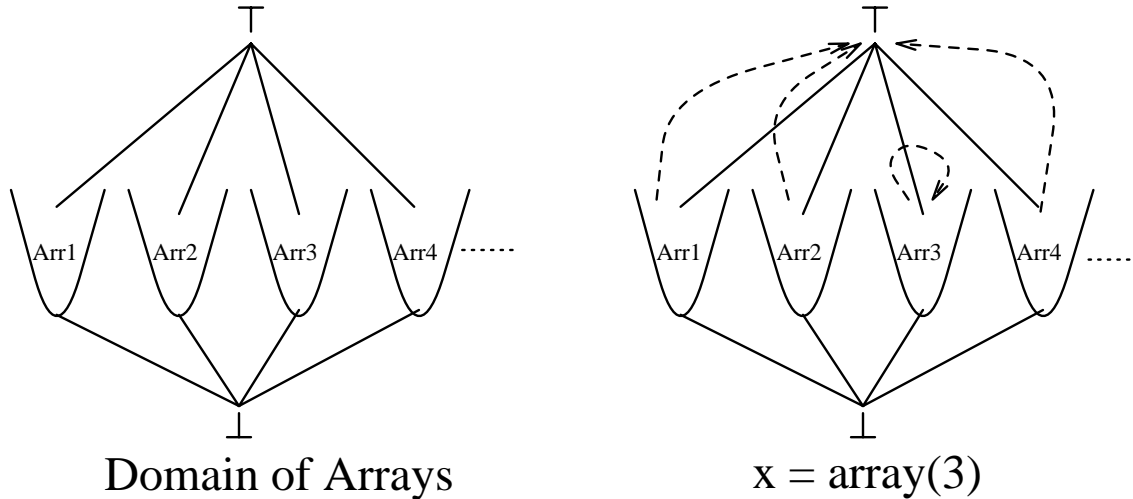


Figure 3: The Domain  $V$  and a Closure Operator on  $V$

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

The key observation that provides the foundation of the denotational semantics is that sets corresponding to solutions of constraints in our language can be modeled as retractions of  $V$  - that is, they can be described as solutions of fixpoint equations of the form  $x = f(x)$  where  $f : V \rightarrow V$  is continuous and idempotent. Moreover, the retraction mappings  $f$  satisfy a very special property - they are *extensive* functions in the sense that  $x \sqsubseteq f(x)$ . Intuitively,  $f$  maps each element  $x$  in the domain to a point higher than  $x$ . Extensivity is distinct from monotonicity - one does not imply the other. Functions that are continuous, extensive and idempotent are called continuous *closure operators* [?]. As an example, consider the definition  $\mathbf{x} = \mathbf{array}(3)$ . The elements of  $V$  that satisfy the constraint are easily seen to be solutions of the equation  $x = (\lambda u.u \sqcup [\perp, \perp, \perp])x$ .

A pictorial representation of this function is shown in Figure ?? - it maps  $\perp$  to  $[\perp, \perp, \perp]$ , the least defined array of length 3, it maps  $\top$  and all arrays of length 3 to themselves, and it maps all other values in  $V$  (such as basic values and arrays of length other than 3) to  $\top$ . We leave it to the reader to verify that the function

$(\lambda u.u \sqcup [\perp, \perp, \perp])$  is a closure operator.

There is a deep connection between the operational semantics and the fact that solutions of constraints in our language can be described as fixpoints of closure operators. The basic mechanism by which constraints get imposed in Cid is through unification. Each time unification is performed, new constraints are imposed on some variables. This always adds information; thus the imposition of a constraint can be described via a function that adds to the “information content” of its argument. Such functions are obviously *extensive* functions. Clearly, imposing a constraint twice is no different from imposing it once. Therefore, functions modeling imposition of constraints should be idempotent. Finally, we want the functions to be monotonic and continuous as well since the process of generating constraints is computable. For future reference, we write down some facts about closure operators.

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

In the denotational semantics, definitions will be interpreted as closure operators of type  $ENV \rightarrow ENV$  and expressions will be interpreted as closure operators of type  $(ENV \times V) \rightarrow (ENV \times V)$ . The intuition behind these types is the following. The execution of a definition of the form  $\mathbf{x} = \mathbf{e}$  imposes constraints on  $\mathbf{x}$ , in addition to constraints that are imposed by the evaluation of  $\mathbf{e}$ . Therefore, it is natural to model a definition as a closure operator that take an environment as input and returns a more defined environment that incorporates the effect of all constraints in the definition. The type of expressions is more subtle. Looking back at our discussion of the meaning of the definition  $\mathbf{x} = \mathbf{array}(3)$ , we observe that the expression  $\mathbf{array}(3)$  can be interpreted as the function  $\lambda u.u \sqcup [\perp, \perp, \perp]$ . Thus,  $\mathbf{array}(3)$  is a function of type  $V \rightarrow V$  which, given a value, refines it to a more defined value that is either  $\top$  or an array of length 3. More generally, we must give meaning to an expression of the form  $\mathbf{array}(n)$ ; so we let the function take an additional argument, the environment, so that it can look up values of identifiers, such as  $n$ . Since we wish model this function as a closure operator, we can make it return its input environment as a result - then, the meaning of the expression is a closure operator of type  $(ENV \times V) \rightarrow (ENV \times V)$ . A further degree of generality is required for expressions of the form  $\mathbf{x} = \mathbf{array}(\mathbf{e})$  where  $\mathbf{e}$  is a Cid expression that may impose constraints on the environment — the type of the closure operator is the same, but the returned environment will, in general, be more defined than the input environment.

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

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

**Lemma 4** *If  $f : V \rightarrow V$  and  $g : V \rightarrow V$  are closure operators, any solution to the system of simultaneous equations*

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

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

This lemma lets us compute solutions of a set of fixpoint equations involving closure operators, and talk meaningfully about the least solution of such a set of equations. Note that if we assume only that  $f$  and  $g$  are continuous functions, the system of equations in Lemma ?? need not have a solution, let alone a least solution. In abstract terms, intersection of sets is modeled using composition of closure operators. An interesting fact is that if  $f$  and  $g$  are closure operators, then  $f \circ g$  is not necessarily a closure operator, but  $\bigsqcup (f \circ g)^n$  is a closure operator with the same fixpoints as  $f \circ g$ . An operational interpretation of this is that a value satisfying both constraints can be obtained by interleaving executions of  $f$  and  $g$  until ‘steady-state’ is reached. This interpretation has nothing to do with the actual operational semantics as discussed in Section ??, but provides some intuition for the proof of correspondence between the operational and denotational semantics discussed in Section ??.

An informal example will help to clarify the interaction between the parallel composition of definitions in the operational semantics and the least common solutions of closure operators. From the discussion above, the meaning of the definition  $x = y + 3$  is the closure operator  $g$  on  $ENV$  defined by  $\lambda env. env[x \mapsto env(x) \bigsqcup (env(y) + 3)]$ . Similarly, the meaning of the definition  $y = 1$  is the closure operator  $f$  on  $ENV$  defined by  $\lambda env. env[y \mapsto env(y) \bigsqcup 1]$ . Consider the two definitions together:  $x = y+3$ ;  $y = 1$ . Let the meaning of the two definitions together be denoted by  $h$ . Let  $env_{\perp}$  be the environment in which all identifiers are undefined. Then, from lemma ??,  $h env_{\perp}$  is the limit of the sequence

$$env_{\perp}, f(g(env_{\perp})), f(g(f(g(env_{\perp}))))$$

which evaluates to

$$env_{\perp}, env_{\perp}[y \mapsto 1], env_{\perp}[x \mapsto 4, y \mapsto 1], env_{\perp}[x \mapsto 4, y \mapsto 1], \dots$$

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

The rest of this section develops these ideas more formally. In Section ??, we give a formal definition of the domain  $V$  which was defined informally above. This section can be omitted without loss of continuity. In Section ??, we give the denotational semantics of Cid.

## 5.2 The Semantic Domain

To define the domain of arrays formally, we use a standard construction for defining a domain of (possibly infinite) terms in logic programming, as described in Lloyd's book [?]. First we need some notation. Let  $\omega$  be the set of natural numbers. We use  $\omega^*$  for the set of finite sequences of natural numbers. A sequence is written  $[i_1, \dots, i_n]$ . 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 written  $|X|$  and the size of a sequence  $s$  is written  $|s|$ .

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

1.  $\forall s \in \omega^*$  and  $\forall i, j \in \omega$  we have  $([s, i] \in T \wedge j < i) \Rightarrow (s \in T \wedge [s, j] \in T)$ .
2.  $|\{i \mid [s, i] \in T\}|$  is finite for all  $s \in T$ .

These define finitely branching trees that may be infinitely deeply nested. The sequences are the tree addresses of the nodes of the tree. We define  $br(s, t)$  to be the number of successors of the node  $s$  in the tree  $t$ ; if the tree is clear from context we will write  $br(s)$ . If this number is 0 we have a leaf.

The domain  $V$  is defined in two stages. First we define a domain  $W$  and then we add a top element, written  $\top$ . The domain  $W$  is defined as follows. Let  $Atom$  be a given domain of atomic values and let  $Arrays$  be the set of array constructors written in infix form as  $\{[ ], [ ], [ , ], \dots\}$  or for ease of reference as  $\{array_1, array_2, \dots\}$ . Let  $A = Atom \cup \{\Omega\} \cup Arrays$  where  $\Omega$  stands for the undefined element.

**Definition 4** An element of  $W$  is a function  $f : t \rightarrow A$  where  $t$  is a non-empty tree. The function  $f$  satisfies  $\forall s \in t. br(s) = 0 \Rightarrow f(s) \in (Atom \cup \Omega) \wedge br(s) = n \neq 0 \Rightarrow f(s) = array_n$ . The ordering between elements of  $W$  is defined as follows:  $f \sqsubseteq g$  iff

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

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

### 5.3 The Denotational Semantics of Cid

In defining the denotational semantics, we need environments that assign values in  $V$  to identifiers. Let  $ENV$ , the domain of these semantic environments, be  $Id \rightarrow V$ ; we will use  $env$  to refer to any element of this domain. The environment in which all identifiers are mapped to  $\top$  is called  $env_{\top}$ . The semantic function  $\mathcal{C}$  interprets definitions as closure operators of type  $ENV \rightarrow ENV$  and the semantic function  $\mathcal{E}$  interprets expressions as closure operators of type  $(ENV \times V) \rightarrow (ENV \times V)$ . We assume that the environment contains distinguished bindings for each of the top-level functions permitted in a Cid program. Strictly speaking then, the type of  $ENV$  must be modified so that an environment is the sum of an ordinary environment as described above and an environment in which the values of functions can be looked up. We will not make this explicit in the semantic clauses below except when we define the meaning of the function expression. Since we are only looking at the first-order case, 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  $\langle env, v \rangle$  returned by the denotation of an expression is formed with the a special pair constructor, written  $\langle , \rangle$ , which is strict with respect to  $\top$  i.e.  $(\forall env, a)[\langle env, \top \rangle = \top \wedge \langle env_{\top}, a \rangle = \top]$ . We use the notation **lcs** to stand for the *least common solution* of a set of equations. 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  $x = a \sqcup x$ .

The semantic clauses are shown in Figure ??.

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  $V \times V$ , i.e. an element of  $(V \times V) \rightarrow_C (V \times V)$ . This function is defined below using a least fixed point operator, written  $\mu$ , on the space of closure operators on  $V \times V$ . We assume that the symbol  $F$  is bound to  $\mathcal{F}[[F]]$  in all the environments used in the definition of  $\mathcal{E}$ .

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

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

$$\begin{aligned}
\mathcal{C}[[x = e]] \text{ env } &= \text{lcs} \left\{ \begin{array}{l} \text{env} \sqsubseteq \text{env}' \\ \text{env}', r = \mathcal{E}[[e]] \text{ env}' \ r \\ \text{env}'[x] = r \end{array} \right. \\
&\quad \text{in if } r = \top \text{ then } \text{env}_\top \text{ else } \text{env}' \\
\mathcal{C}[[\text{def}_1 ; \text{def}_2]] \text{ env } &= \text{lcs} \left\{ \begin{array}{l} \text{env} \sqsubseteq \text{env}' \\ \text{env}' = \mathcal{C}[[\text{def}_1]] \text{ env}' \\ \text{env}' = \mathcal{C}[[\text{def}_2]] \text{ env}' \end{array} \right. \\
&\quad \text{in } \text{env}' \\
\mathcal{E}[[\text{const}]] \text{ env } \ a &= \langle \text{env}, \mathcal{K}(\text{const}) \sqcup a \rangle \\
\mathcal{E}[[x]] \text{ env } \ a &= \langle \text{env}[x \mapsto (\text{env}(x) \sqcup a)], \text{env}(x) \sqcup a \rangle \\
\mathcal{E}[[e_1 \text{ op } e_2]] \text{ env } \ a &= \text{lcs} \left\{ \begin{array}{l} \text{env} \sqsubseteq \text{env}' \\ \text{env}', v_1 = \mathcal{E}[[e_1]] \text{ env}' \ v_1 \\ \text{env}', v_2 = \mathcal{E}[[e_2]] \text{ env}' \ v_2 \\ r = (v_1 \text{ op } v_2) \sqcup a \end{array} \right. \\
&\quad \text{in } \langle \text{env}', r \rangle \\
\mathcal{E}[[\text{array}(e)]] \text{ env } \ a &= \text{lcs} \left\{ \begin{array}{l} \text{env} \sqsubseteq \text{env}' \\ \text{env}', n = \mathcal{E}[[e]] \text{ env}' \ n \\ r = \text{Array}(n) \sqcup a \end{array} \right. \\
&\quad \text{in } \langle \text{env}', r \rangle \\
\mathcal{E}[[[L_1, L_2, \dots, L_n]]] \text{ env } \ a &= \text{lcs} \left\{ \begin{array}{l} \text{env} \sqsubseteq \text{env}' \\ a \sqsubseteq r \\ \text{env}'[L_1] = r[1] \\ \dots \\ \text{env}'[L_n] = r[n] \end{array} \right. \\
&\quad \text{in } \langle \text{env}', r \rangle \\
\mathcal{E}[[\text{cond}(e_1, e_2, e_3)]] \text{ env } \ a &= \text{env}', b = \mathcal{E}[[e_1]] \text{ env}' \ b \\
&\quad \text{in if } b \text{ then } \mathcal{E}[[e_2]] \text{ env}' \ a \\
&\quad \quad \text{else } \mathcal{E}[[e_3]] \text{ env}' \ a \\
\mathcal{E}[[e_1[e_2]]] \text{ env } \ a &= \text{lcs} \left\{ \begin{array}{l} \text{env} \sqsubseteq \text{env}' \\ a \sqsubseteq r \\ \text{env}', v_1 = \mathcal{E}[[e_1]] \text{ env}' \ v_1 \\ \text{env}', v_2 = \mathcal{E}[[e_2]] \text{ env}' \ v_2 \\ v_1[v_2] = r \end{array} \right. \\
&\quad \text{in } \langle \text{env}', r \rangle \\
\mathcal{E}[[F(e)]] \text{ env } \ a &= \text{lcs} \left\{ \begin{array}{l} \text{env} \sqsubseteq \text{env}' \\ a \sqsubseteq r \\ \langle \text{env}', v \rangle = \mathcal{E}[[e]] \text{ env}' \ v \\ v, r = \mathcal{E}_F[[F]] \ v \ r \end{array} \right. \\
&\quad \text{in } \langle \text{env}', r \rangle
\end{aligned}$$

Figure 4: Denotational Semantics of Cid

needs to be related to the operational semantics in order to verify that these abstract semantic descriptions really do correspond to the execution of Cid programs. We do this in the next section.

## 6 Relating the Semantic Definitions

In this section we an informal sketch of the proof that the denotational semantics and the operational semantics coincide for our notion of observations. Because of the length and density of these proofs, we leave the details to the appendices.

We first describe informally our notion of observation. In the denotational semantics, we have infinite objects such as infinitely deeply nested arrays, but we will take the point of view that we can only make “finite observations”. Intuitively, this corresponds to observing the bindings of variables only upto finite depth. The denotational semantics can be viewed as *collecting the result of infinitely many finite observations* but the primitive observations themselves are finite, in the above sense.

**Definition 5** *Let  $\langle D, e, \rho, F \rangle$  be a configuration that is being reduced. A basic observation about  $\langle D, e, \rho, F \rangle$ , is a pair  $\langle x, v \rangle$  where  $x$  is a variable and  $v$  is a finite element of  $V$  such that  $x$  is bound to a value that is greater than  $v$  in  $\rho$ . An observation is a conjunction of a finite set of basic observations. Also, if  $e$  is a basic value, we regard finite approximants to the value that results as being basic observables.*

We distinguish processes on the basis of the observations that can be made at any finite stage of the computation, but our semantics does not say anything about termination of processes. Thus, a process that imposes a constraint is not distinguished from one that imposes the same constraint but has a divergent subcomputation proceeding silently — the “least” program is `let x = x in x` and so is the program that imposes no constraint but diverges. We do however, view *Error* as a finite value and hence observable. We view two programs as observably identical if one can make exactly the same observations of them *in all contexts*. The main result of this section is the full abstraction result; namely, that two programs are observably identical if and only if they have the same denotation.

### Operational properties

To proceed with the proofs, we need some basic facts about the operational semantics, specifically we would like a determinacy theorem or Church-Rosser theorem. Some of these properties were discussed in Section ?? on the operational semantics.

The first important property is *operational monotonicity*. This says that if a configuration  $\langle D, e, \rho, F \rangle$  has an enabled transition and  $\rho'$  is a syntactic environment that dominates  $\rho$  then  $\langle D, e, \rho', F \rangle$  has “essentially” the same transition enabled. Thus enhancing the inputs cannot disable an enable transition.

This can be proved by a routine structural induction once the notion of “essentially the same transition” is spelled out.

The second important property is *local confluence*. We say that two transitions *commute* if whenever they are both enabled in a configuration then they do not “interfere”. More precisely, suppose that  $\langle D, e, \rho, F \rangle$  can take a step and become  $\langle D_1, e_1, \rho_1, F_1 \rangle$  and also could take a different step and become  $\langle D_2, e_2, \rho_2, F_2 \rangle$ . Then it is possible for  $\langle D_1, e_1, \rho_1, F_1 \rangle$  to take a step and become  $\langle D_3, e_3, \rho_3, F_3 \rangle$  and also for  $\langle D_2, e_2, \rho_2, F_2 \rangle$  to take a step and become  $\langle D_3, e_3, \rho_3, F_3 \rangle$ . The idea is that if there are multiple transitions they come from independent concurrently executing subagents and not from internal indeterminate choices. In the operational semantics, we have shown that Cid has the property that any two transitions enabled in a configuration commute. From this one can prove the general confluence property by “pasting together diamonds” as in any proof of the Church Rosser theorem for the lambda calculus [?].

## 6.1 Reduction Preserves Meaning

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

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  $ENV \times V$ . We use the semantic functions  $\mathcal{E}, \mathcal{F}$  and  $\mathcal{C}$  defined previously and the same notational conventions.

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

We require that the semantic environment  $\text{env}$  and the syntactic environment  $\rho$  satisfy

$$\text{Dom}(\text{env}) \cap FL = \emptyset \dots \star$$

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  $\rho$ ) 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.

Appendix ?? spells the proof out in detail.

## 6.2 The Adequacy Theorem

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

We first define a relationship  $\preceq$  between syntactic expressions,  $e$ , and closure operators,  $f$ , on  $ENV \times V$ . The main theorem proves that for all syntactic expressions  $e$ ,  $\mathcal{E}[[e]] \preceq e$ . Intuitively,  $\mathcal{E}[[e]] \preceq e$  means that given any finite approximant to the result predicted by  $\mathcal{E}[[e]]$ , there is a finite sequence of reductions evaluating  $e$  in a suitable syntactic environment, that produces a more refined value. In particular, if the result predicted by  $\mathcal{E}[[e]]$  is  $\top$ , evaluating  $e$  in a suitable syntactic environment results in *error*. The proof that  $\mathcal{E}[[e]] \preceq e$ , for all expressions  $e$  proceeds by structural induction on the expressions. The heart of the proof is a sequence of lemmas corresponding to the cases of the structural induction.

The difficult part of any 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. For most of the constructs in the language it is clear how one ought to proceed. The subtle case is when one has parallel imposition of constraints. We make use of the fact that the semantic prescription for determining the least common fixed point of a pair of closure operators suggests an interleaving of the reduction sequences of the subterms. This is the key to the whole adequacy proof.

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

## Full abstraction

In full abstraction we aim to establish that the denotational semantics is an accurate guide to program behavior 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 behavior of syntactic expressions as follows.

**Definition 6**  $e_1 \sqsubseteq_{op} e_2$  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 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 2** *The denotational semantics is inequationally adequate i.e*

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

The proof is given in the appendix.

The equivalence of the two orders, the operationally defined order and the information order of the denotational semantics, is full abstraction. It is essentially a consequence of the fact that all the prime elements of the space of the closure operators on  $ENV \times V$  are expressible as the meanings of expressions. As in Plotkin's proof of full abstraction for PCF [?], the crux of the proof is the construction of contexts that can semantically distinguish two different expressions. Since we do not have higher types it suffices to work with the prime elements rather than the finite elements.

**Theorem 3** *The denotational semantics is fully-abstract i.e.*

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

## 7 Conclusions

In this paper, we defined Cid, a first-order functional language with logic variables, and gave it a Plotkin-style structured operational semantics. The operational semantics incorporated an explicit treatment of aliasing

introduced by unification, and the interleaving of computations in different parts of the program. While such features usually complicate the denotational model, we showed that Cid programs could be given a simple denotational treatment couched entirely in terms of equations and equation solving. The connection between equation solving and the operational semantics was established through a novel use of closure operators on Scott domains. Finally, we showed that the denotational semantics was fully abstract with respect to the operational semantics.

This exercise in formal language definition has been useful in helping us understand the subtleties of introducing logic variables into functional languages. One such subtlety is *referential transparency*. In functional languages, referential transparency is interpreted as substitutivity of identities — given a definition  $x = e$ , uses of  $x$  can be replaced with  $e$  without changing the meaning of the program. Part of the appeal of functional languages arises from the fact that they are referentially transparent. Cid is not referentially transparent in that sense — for example, given the definition  $x = \text{array}(5)$ , we cannot replace occurrences of  $x$  in the program with the expression  $\text{array}(5)$ . The constraint-oriented semantics developed here suggest that the addition of logic variables to a functional language results in a shift of paradigm; programs should be seen as manipulations of *objects* rather than of values as is the case for pure functional languages. Thus, the expression  $\text{array}(5)$  creates an array object, and definitions bind names to objects. Since objects are created by constraint intersection, object identity is important, so we cannot replace occurrences of the name  $x$  with the expression that creates the object bound to it. Although we were, at first, disturbed by the loss of referential transparency, it appears that the functional notion of referential transparency, first used by Quine [?], is not the only definition of this concept in the literature. Whitehead and Russell give a slightly different interpretation to this term — a statement is said to be referentially transparent if its meaning is independent of the context of its assertion [?]. From this perspective, referential transparency is a property of *definitions*, not expressions, and the definition  $x = \text{array}(5)$  is referentially transparent since it is interpreted as an assertion that  $x$  is an element of the set of arrays of length 5. In that case, it is not meaningful to interpret the  $=$  symbol as an operator that binds identifiers to values.

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

of various kinds [?], but we believe that the development of a clean syntax for this class of languages is still an open issue.

Although we have focused on arrays in this paper, the semantic techniques developed here can be applied to any language in which objects are created through constraint intersection. The case of deterministic constraint programming languages is straight-forward since they incorporate only AND-parallelism, and the techniques of this paper can be applied directly. We have recently shown that the techniques of this paper can be used to give semantics to a *higher-order* functional language with logic variables [?].

In a more general context, the full abstraction result in this paper should be of interest to the semantics community, which is actively studying full abstraction [?, ?, ?] for functional languages. Full abstraction for functional languages was first studied by Plotkin in his seminal paper on LCF where full abstraction was obtained by adding parallel-or construct to the language [?]. Intuitively, the parallel-or operator allowed the syntactic definition of the semantic least upper bound function. The semantics of Cid is fully abstract without the addition of new syntactic constructs because the least upper bound function is introduced implicitly into the language through composition of definitions. This can be viewed as semantic evidence of the need for parallelism in the operational model of the language.

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

The extension of this work to the non-deterministic setting is still in a state of flux [?, ?]. The semantic accounts of these papers are fully abstract for reasonable notions of observation. However, the semantics in both these papers explicitly encode operational notions, which makes reasoning in the abstract semantics non-trivial and complex. It remains an open question if the simple view of functional languages with logic variables achieved in this paper can be extended to programs with indeterminacy.

*Acknowledgments:* We thank Arvind, Rishiyur Nikhil, Gary Lindstrom, Albert Meyer and Jean-Louis Lassez for useful conversations.

## References

- [1] Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11, October 1989.
- [2] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.
- [3] H. Barendregt and M. van Leeuwen. functional programming and the language Tale. Technical report, Mathematical Institute, Utrecht, Netherlands, 1985.
- [4] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic. North-Holland, Amsterdam, revised edition edition, 1984.
- [5] G. Berry, P. L. Curien, and J. J. Levy. Full abstraction for sequential languages; the state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 3, pages 89–132. Cambridge University Press, 1985.
- [6] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The Simple code. Technical Report UCID-17715, Lawrence Livermore Laboratory, 1978.
- [7] J-Y. Girard. personal communication, 1987.
- [8] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, London, 1985.
- [9] J. Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.
- [10] R. Jagadeesan and K. Pingali. An abstract semantics for a higher-order functional language with logic variables. Technical Report to appear, Cornell University, 1991.
- [11] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475, 1974.
- [12] G. Lindstrom. Functional programming and the logical variable. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, 1985.
- [13] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.
- [14] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Publishing Company, 1981.

- [15] S. Mantha, G. Lindstrom, and L. George. A semantic framework for functional programming with constraints. Unpublished Technical Report.
- [16] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [17] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–23, 1977.
- [18] K. Mulmuley. *Full Abstraction and Semantic Equivalence*. ACM Distinguished Dissertation Series. MIT Press, 1987. CMU Ph.D. dissertation.
- [19] R. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical Report CSG Memo 265, M.I.T. Laboratory for Computer Science, 1986.
- [20] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.
- [21] Gordon Plotkin. LCF considered a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.
- [22] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.
- [23] W. Quine. *Word and Object*. M.I.T. Press, 1960.
- [24] Y. Lichtenstein R. Gerth, M. Codish and E. Shapiro. Fully-abstract denotational semantics for flat concurrent prolog. In *Proceedings of IEEE Conference on Logic in Computer Science*, pages 320–335, 1988.
- [25] J. Parrow R. Milner and D. Walker. Mobile processes. Technical report, University of Edinburgh, 1989. Draft.
- [26] U. Reddy. *Logic Programming — Functions, Relations and Equations*, chapter On the relationship between logic and funtional languages. Prentice-Hall, 1986.
- [27] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the conference on Principles of Programming Languages*, 1991.
- [28] M. Sato and T. Sakurai. QUTE: a functional language based on unification. In *Logic Programming: functions, relations and equations*, 1986.
- [29] D. Scott. Data types as lattices. *SIAM Journal of Computing*, 1976.

- [30] E. Shapiro. A subset of concurrent Prolog and its interpreter. Technical Report TR-003, Institute for new generation computer technology, 1983.
- [31] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.
- [32] M. B. Smythe. Powerdomains. *Journal of Computer and System Sciences*, 16:23–36, 1978.
- [33] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 1986.
- [34] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, 1977.
- [35] D. A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9:31–49, 1979.
- [36] K. Ueda. Guarded horn clauses. Technical Report 103, ICOT, Tokyo, Japan, 1985.
- [37] A.N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, England, 1910.

## A One-step Reduction Preserves Meaning

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

A syntactic environment  $\rho$  is a collection of alias sets and each alias set is a set consisting, in general, of identifiers and terms. Suppose that  $\rho$  is a syntactic environment, we shall write  $EQ(\rho)$  for the set of equations generated from  $\rho$ . We define  $EQ(\rho)$  as the reflexive, transitive and symmetric closure of the union of the equations generated from each alias set  $A_1, A_2, \dots$  in  $\rho$ . 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  $A = \{x_1, \dots, x_N\}$ . Then  $EQ(A) = \{x_1 = x_2, x_1 = x_3, \dots, x_2 = x_3, \dots\}$ . 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  $\{x_1 = c, x_2 = c, \dots\}$ . In case (iii) we have the possibility of an inconsistency. If we have an inconsistent alias set  $A$ , and  $\{x_1, \dots, x_N\}$  are all the identifiers in  $A$  then  $EQ(A) = \{x_1 = \top, x_2 = \top, \dots, x_N = \top\}$ . If we have a consistent alias set, then the assumptions of case (iii) require that the terms must all be arrays of the same size or identifiers. For simplicity we consider the case where there are two arrays of size two and no identifiers. If  $A = \{[L1, L2], [L3, L4]\}$  then we set  $EQ(A) = \{L1 = L3, L2 = L4\}$ . If we have identifiers, say  $x$  and  $y$  in  $A$  as well, we add the equations  $x = y, x = [L1, L2], y = [L1, L2], x = [L3, L4], y = [L3, L4]$  to  $EQ(A)$ . If the equations induced by equating array components involve two arrays then the resulting equations are also added to  $EQ(A)$ . Thus  $EQ(A)$  may contain infinitely many equations. It should be clear that  $EQ(A)$  is defined to express all the semantic consequences of a given set of equations and is not intended to be an effective procedure. The 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  $\overset{*}{\rightsquigarrow}$  preserves the meaning.

**Lemma 5** *If  $\rho \overset{*}{\rightsquigarrow} \rho'$  then  $EQ(\rho) = EQ(\rho')$ .*

*Proof:* We know, by theorem 1, that the sequence of  $\rightsquigarrow$  steps terminates, thus we need only show that if  $\rho \rightsquigarrow \rho'$  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  $ENV \times V$ . We use the semantic functions  $\mathcal{E}$ ,  $\mathcal{F}$  and  $\mathcal{C}$  defined previously and the same notational conventions.

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

We require that the semantic environment  $\text{env}$  and the syntactic environment  $\rho$  satisfy

$$\text{Dom}(\text{env}) \cap FL = \emptyset \dots \star$$

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  $\rho$ ) 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  $\rho$ .

**Theorem 4** *Suppose that the following rewrite is possible:*

$$\langle D, e, \rho, FL \rangle \rightarrow \langle D', e', \rho', FL' \rangle$$

*then  $\forall \text{env}$  satisfying the condition  $\star$  with respect to both  $\rho$  and  $\rho'$  and  $\forall a \in V$   $\mathcal{M}[\langle D, x, \rho, FL \rangle] \text{ env } a|_{bv(\rho)} = \text{env } a|_{bv(\rho)}$ , if and only if*

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

*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.

$$\langle D, x, \rho, FL \rangle \rightarrow \langle D, \rho[x], \rho, FL \rangle$$

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

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

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

$$\begin{aligned} \text{env}' &= \mathcal{C}[D \cup \rho] \text{ env}' \\ \text{env}', a &= \mathcal{E}[x] \text{ env}' \ a \end{aligned}$$

Assuming that  $x = e \in EQ(D \cup \rho)$ , our aim is to prove that the above is equivalent to

$$\begin{aligned} \text{env}' &= \mathcal{C}[D \cup \rho] \text{ env}' \\ \text{env}', a &= \mathcal{E}[e] \text{ env}' \ a \end{aligned}$$

This reduces to proving that

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

and

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

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

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

So  $\mathcal{C}[x = e] \text{ env}' = \text{env}'$  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  $env', r = \langle env'[x \mapsto env'[x] \sqcup r], env'[x] \sqcup r \rangle$ . Hence, we have  $env'[x] \sqcup r = r = env'[x]$ . 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.

$$\text{Basic Operations: } \frac{\langle D, e_1, \rho, FL \rangle \rightarrow \langle D^*, e_1^*, \rho^*, FL^* \rangle}{\langle D, e_1 \ op \ e_2, \rho, FL \rangle \rightarrow \langle D^*, e_1^* \ op \ e_2, \rho^*, FL^* \rangle}$$

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

$$\mathcal{M}[\langle D, e_1 \ op \ e_2, \rho, FL \rangle] \ env \ a = \begin{cases} env \sqsubseteq env' & 1 \\ a \sqsubseteq r' & 2 \\ env' = \mathcal{C}[D \cup \rho] \ env' & 3 \\ env', v'_1 = \mathcal{E}[e_1] \ env' \ v'_1 & 4 \\ env', v'_2 = \mathcal{E}[e_2] \ env' \ v'_2 & 5 \\ r' = (v'_1 \ op \ v'_2) \sqcup a & 6 \end{cases} \text{ lcs} \\ \text{in } \langle env', r' \rangle$$

The meaning of the configuration after the rewrite is

$$\mathcal{M}[\langle D, e_1^* \ op \ e_2, \rho, FL \rangle] \ env \ a = \begin{cases} env \sqsubseteq env'' & 7 \\ a \sqsubseteq r'' & 8 \\ env'' = \mathcal{C}[D^* \cup \rho^*] \ env'' & 9 \\ env'', v''_1 = \mathcal{E}[e_1^*] \ env'' \ v''_1 & 10 \\ env'', v''_2 = \mathcal{E}[e_2] \ env'' \ v''_2 & 11 \\ r'' = (v''_1 \ op \ v''_2) \sqcup a & 12 \end{cases} \text{ lcs} \\ \text{in } \langle env'', r'' \rangle$$

The two systems of equations are identical except for the fact that we have  $D^*$  instead of  $D$  and similarly for  $e_1$  and  $\rho$ . 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  $\top$ .

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

$$\text{Array: } \begin{array}{l} 1. \frac{\langle D, e, \rho, FL \rangle \rightarrow \langle D^*, e^*, \rho^*, FL^* \rangle}{\langle D, \text{array}(e), \rho, FL \rangle \rightarrow \langle D^*, \text{array}(e^*), \rho^*, FL^* \rangle} \\ 2. \langle D, \text{array}(n), \rho, FL \rangle \rightarrow \langle D, [L1, \dots, Ln], \rho^*, FL^* \rangle \\ \text{where } L1, \dots, Ln \in FL \\ \rho^* = \rho \cup \{\{L1\}, \dots, \{Ln\}\} \\ FL^* = FL - \{L1, \dots, Ln\} \end{array}$$

The proof for the first transition follows from the induction hypothesis on the operational transition

$$\langle D, e, \rho, FL \rangle \rightarrow \langle D^*, e^*, \rho^*, FL^* \rangle.$$

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

$$\mathcal{M}[\langle D, \text{array}(n), \rho, FL \rangle] \text{ env } a = \begin{array}{l} \text{lcs} \left\{ \begin{array}{ll} \text{env} \sqsubseteq \text{env}' & 1 \\ a \sqsubseteq r' & 2 \\ \text{env}' = \mathcal{C}[\langle D \cup \rho \rangle] \text{ env}' & 3 \\ r' = \text{Array}(K(n)) \sqcup a & 4 \end{array} \right. \\ \text{in } \langle \text{env}', r' \rangle \end{array}$$

and

$$\mathcal{M}[\langle D, \text{array}(n), \rho, FL \rangle] \text{ env } a = \begin{array}{l} \text{lcs} \left\{ \begin{array}{ll} \text{env} \sqsubseteq \text{env}'' & 5 \\ a \sqsubseteq r'' & 6 \\ \text{env}'' = \mathcal{C}[\langle D \cup \rho \rangle] \text{ env}'' & 7 \\ \text{env}''[L1] = r'[1] & 8 \\ \dots & \\ \text{env}''[Ln] = r'[n] & 9 \end{array} \right. \\ \text{in } \langle \text{env}'', r'' \rangle \end{array}$$

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  $\text{env}'$  and  $\text{env}''$  will differ in that the former will have no bindings for the identifiers  $L1, \dots, Ln$  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:      1.  $\langle D, F(e), \rho, FL \rangle \rightarrow \langle D^*, e_1, \rho^*, FL^* \rangle$   
                           where  $D^* = D \cup \{x = e\} \cup \text{defs}_F[x/\text{arg}_F][y/\text{local}_F](x, y \in FL)$   
                            $e_1 = \text{exp}_F[x/\text{arg}][y/\text{local}_F]$   
                            $\rho^* = \rho \cup \{\{x\}, \{y\}\}$   
                            $FL^* = FL - \{x, y\}$

The meanings of the configurations are given by

$$\mathcal{M}[\langle D, F(e), \rho, FL \rangle] \text{ env } a = \begin{cases} env \sqsubseteq env' & 1 \\ a \sqsubseteq r' & 2 \\ env' = \mathcal{C}[\langle D \cup \rho \rangle] \text{ env}' & 3 \\ \langle env', v' \rangle = \mathcal{E}[e] \text{ env}' \text{ } v' & 4 \\ \langle v', r' \rangle = \mathcal{F}[F] v' r' & 5 \end{cases} \text{ in } \langle env', r' \rangle$$

and

$$\mathcal{M}[\langle D^*, e_1, \rho^*, FL^* \rangle] \text{ env } a = \begin{cases} env \sqsubseteq env'' & 6 \\ a \sqsubseteq r'' & 7 \\ env'' = \mathcal{C}[\langle D^* \cup \rho^* \rangle] \text{ env}'' & 8 \\ \langle env'', r'' \rangle = \mathcal{E}[e_1] \text{ env}'' \text{ } r'' & 9 \end{cases} \text{ in } \langle env'', r'' \rangle$$

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

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

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:

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

$$2. \langle D \cup \{x = y\}, e, \rho, FL \rangle \rightarrow \langle D, e, \mathcal{U}(\rho, \{x, y\}), FL \rangle$$

(if  $\mathcal{U}(\rho, \{x, y\})$  is consistent)

$$\langle D \cup \{x = y\}, e, \rho, FL \rangle \rightarrow Error \text{ (otherwise)}$$

$$3. \langle D \cup \{x = c\}, e, \rho, FL \rangle \rightarrow \langle D, e, \mathcal{U}(\rho, \{x, c\}), FL \rangle$$

(if  $\mathcal{U}(\rho, \{x, c\})$  is consistent)

$$\langle D \cup \{x = c\}, e, \rho, FL \rangle \rightarrow Error \text{ (otherwise)}$$

$$4. \langle D \cup \{x = [L1, \dots, Ln]\}, e, \rho, FL \rangle \rightarrow \langle D, e, \mathcal{U}(\rho, \{x, [L1, \dots, Ln]\}), FL \rangle$$

(if  $\mathcal{U}(\rho, \{x, [L1, \dots, Ln]\})$  is consistent)

$$\langle D \cup \{x = [L1, \dots, Ln]\}, e, \rho, FL \rangle \rightarrow Error \text{ (otherwise)}$$

The reasoning is quite straightforward now. There are four sub-cases to consider, corresponding to the four rules above. The first case follows immediately from the inductive hypothesis. In the second case, we add the equation  $x = y$  to  $\rho$  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 constraints present in computing the meaning of the original configuration and setting the meaning to  $\top$  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  $\rho$ . ■

## B Adequacy of Denotational Semantics

In this section, we prove that the operational semantics actually attains the values predicted by the denotational semantics. Along with the fact that one-step reduction preserves meaning, this means that the results predicted by the operational and denotational semantics match exactly; this is usually called *adequacy*. Since infinite objects are present in the semantic domain, we cannot claim that every output predicted by the denotational semantics is actually attained by a *finite* reduction sequence. Instead, we show that any *finite approximant* of the predicted value can be produced by a finite reduction sequence. We first define an inclusive predicate  $\preceq$  between syntactic expressions  $e$  and closure operators  $f$  on  $ENV \times V$ . The main theorem of this section proves that for all syntactic expressions  $e$ ,  $\mathcal{E}[[e]] \preceq e$ .  $\mathcal{E}[[e]] \preceq e$  intuitively means that given any finite approximant to the result predicted by  $\mathcal{E}[[e]]$ , there is a finite sequence of reductions evaluating  $e$  in a suitable syntactic environment, that produces a more refined value. In particular, if the result predicted by  $\mathcal{E}[[e]]$  is  $\top$ , evaluating  $e$  in a suitable syntactic environment results in *error*.

The proof that  $\mathcal{E}[[e]] \preceq e$ , for all expressions  $e$  proceeds by structural induction on the expressions. The inductive argument is an interleaving lemma that constructs the reduction sequence corresponding to a set of semantic equations, assuming that we can construct the reductions corresponding to each equation.

The rest of the section is organised as follows. First, we discuss some operational properties that are useful for the proof. Next, we describe the proof for one inductive case: the case of sets of equations. This subsection highlights the main ideas of the proof. Finally, we present the full proof.

### B.1 Operational facts

We first define a transition relation that is useful in the proof. Intuitively, the relations  $\mapsto_s$  differs from  $\longrightarrow$  in allowing addition of new constraints to the  $D$  (unresolved constraints) component of the operational configurations. Define the transition relation  $\mapsto_s$  as follows.

- $\langle D, e, \rho, FL \rangle \longrightarrow \langle D', e', \rho', FL' \rangle \Rightarrow \langle D, e, \rho, FL \rangle \mapsto_s \langle D', e', \rho', FL' \rangle$
- $\langle D, e, \rho, FL \rangle \mapsto_s \langle D \cup \{x = e'\}, e, \rho, FL \rangle$

$\mapsto_s^*$  is the reflexive and transitive closure of  $\mapsto_s$ .

The transition systems  $\longrightarrow$  and  $\mapsto_s$  are closely related. The following lemma generalizes the Church-Rosser property, given by lemma ???. It can be viewed as saying that addition of new constraints cannot disable enabled reductions.

**Lemma 6** *Let  $conf \mapsto_s^* conf_1 \wedge conf \longrightarrow^* conf_2$ . Then, there exists a configuration  $conf_3$  such that  $conf_2 \mapsto_s^* conf_3 \wedge conf_1 \longrightarrow^* conf_3$ .*

## B.2 Proof for Composition of Definitions

Inclusive predicates are key components in many adequacy proofs [?] [?]. They relate the semantic values with syntactic expressions. They are primarily used to establish that a predicted semantic value is actually attained by rewriting. For defining the inclusive predicate relating expressions and closure operators on  $ENV \times V$ , we need to develop notation that relates syntactic and semantic values as well as syntactic and syntactic environments.

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

**Definition 7** *Syntactic values and environments are related to semantic values and environments as follows:*

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

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

The following lemma states that the relation  $\preceq$  defined above is *inclusive* [?], and satisfies natural monotonicity properties. The proof is immediate and is omitted.

**Lemma 7** *Inclusivity and monotonicity properties of  $\preceq$ :*

1.  $v \preceq (e, \rho) \wedge v' \sqsubseteq v \Rightarrow v' \preceq (e, \rho)$
2.  $v_1 \preceq (x_1, \rho) \wedge v_2 \preceq (x_2, \rho) \wedge \{x_1 = x_2\} \in \rho \Rightarrow v_1 \sqcup v_2 \preceq (x_1, \rho)$
3. Let  $\{v_i | i\}$  be a chain in  $V$ . Let  $e$  be an expression. Then,  $(\forall i) [v_i \preceq (e, \rho)] \Rightarrow \bigsqcup_i \{v_i | i\} \preceq (e, \rho)$ .
4.  $env \preceq \rho \wedge env' \sqsubseteq env \Rightarrow env' \preceq \rho$
5.  $env_1 \preceq \rho \wedge env_2 \preceq \rho \sqsubseteq env \Rightarrow env_1 \sqcup env_2 \preceq \rho$

6. Let  $\{env_i|i\}$  be a chain in  $ENV$ . Then,  $(\forall i) [env_i \preceq \rho] \Rightarrow \bigsqcup_i \{env_i|i\} \preceq \rho$ .

For notational convenience, we follow the convention that the syntactic environment associated with the operational configuration *error* is inconsistent. Furthermore, we denote finite elements of the semantic domains by the subscript  $f$ . For example, a finite element of the value domain will usually be denoted by  $a_f$  or  $b_f$ . A finite element of  $ENV$  will usually be denoted by  $env_f$  and a finite element of  $ENV \times V$  will usually be denoted by  $r_f$ .

Now, we have the machinery to relate finite sets of equations  $E$  of the form  $x = e$  and closure operators  $g$  on  $ENV$ . Roughly speaking,  $g \preceq E$  means that the set of equations imposes more constraints on the environment than the closure operator  $g$ . In the following definition, we use  $\#$  in the expression part of the configuration to indicate that the actual expression is not relevant to the definition. The use of  $\star$  in the environment part and the use of  $\star\star$  in the freelist part of the configuration are motivated by the same reason.

**Definition 8**  $E$  covers  $g$ , written  $g \preceq E$ , is defined as follows. Let

- $env \preceq \rho$
- $g\ env = env'$

Then, given  $\langle E, \#, \star, \star\star \rangle \xrightarrow{*}_s \langle D, \#, \rho, FL \rangle$ ,

$$(\forall env_f \sqsubseteq env') (\exists) [\langle D, \#, \rho, FL \rangle \xrightarrow{*} \langle D', \#, \rho_{res}, FL' \rangle \wedge env_f \preceq \rho_{res}]$$

The interesting case of the above definition is when  $\langle D, \#, \rho, FL \rangle = \langle E, \#, \star, \star\star \rangle$ . The definition is set up in greater generality to enable the proofs to go through smoothly. Consider the case when  $\langle D, \#, \rho, FL \rangle = \langle E, \#, \star, \star\star \rangle$ . Let  $\rho$  be more constrained than  $env$ . Let  $env'$  be the result of applying  $g$  to  $env$ . Then, given any finite approximant  $env_f$  to  $env'$ , there is a way of reducing the equations  $E$  in syntactic environment  $\rho$  for a finite number of steps such that the resulting syntactic environment  $\rho_{res}$  is more constrained than  $env_f$ . In particular, if  $env'$  is the error environment, evaluating  $E$  in  $\rho$  results in *error*.

Now, we prove the case of structural induction corresponding to the case of combining equations. The difficult constituent 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  $g_1$  and  $g_2$  are two closure operators that correspond to the imposition of two constraints given as sets of equations  $E_1$  and  $E_2$ . Suppose that we know how to construct reduction sequences corresponding to  $E_1$  and  $E_2$  individually. Then, since we know that the least common fixed point of  $g_1$  and  $g_2$  is the least fixed point of  $(g_1 \circ g_2)$ , we can construct an interleaved reduction sequence of  $E_1$  and  $E_2$  corresponding to the computing the iterates

of  $(g_1 \circ g_2)$ . In other words, the special form of the fixed point iteration provides guidance about how to construct the interleaved reduction sequence. The proof of the following lemma formalizes this intuition.

**Lemma 8** *Let  $g_1, g_2$  be closure operators on  $ENV$ . Let  $g_1 \preceq E_1$  and  $g_2 \preceq E_2$ . Then,  $g \preceq E_1 \cup E_2$ , where  $g$  is defined as follows:*

$$g \text{ env} = \text{lcs} \begin{cases} \text{env} \sqsubseteq \text{env}' \\ \text{env}' = g_1 \text{ env}' \\ \text{env}' = g_2 \text{ env}' \end{cases} \text{ in } \text{env}'$$

*Proof:* Let  $E = E_1 \cup E_2$ . Let

- $g \text{ env} = \text{env}'$
- $\text{env} \preceq \rho$
- $\langle E, \#, \star, \star\star \rangle \xrightarrow{*}_s \langle D, \#, \rho, FL \rangle$

Note that  $\text{env}' = \bigsqcup_i \{(g_1 \circ g_2)^i \text{ env} | i\}$ . So, if  $\text{env}_f \sqsubseteq \text{env}'$ , there is an  $i$  such that  $\text{env}_f \sqsubseteq (g_1 \circ g_2)^i \text{ env}$ . By induction on  $i$ , we prove that  $\text{env}_f \sqsubseteq (g_1 \circ g_2)^i \text{ env} \Rightarrow (\exists) [\langle D, \#, \rho, FL \rangle \xrightarrow{*} \langle D', \#, \rho_{res}, FL' \rangle]$ , such that  $\text{env}_f \preceq \rho_{res}$ .

**Base:** ( $i = 0$ )

In this case,  $\text{env}_f \sqsubseteq \text{env}$  and the configuration  $\langle D, \#, \rho, FL \rangle$  satisfies required properties.

**Induction:** (assume result for  $i$ )

From the continuity of all functions involved, we deduce the existence of finite environments  $\text{env}_1$  and  $\text{env}_2$  such that,

- $\text{env}_f \sqsubseteq g_1 \text{ env}_1$
- $\text{env}_1 \sqsubseteq g_2 \text{ env}_2$
- $\text{env}_2 \sqsubseteq (g_1 \circ g_2)^i \text{ env}$

From induction hypothesis,  $(\exists) [\langle D, \#, \rho, FL \rangle \xrightarrow{*} \langle D_2, \#, \rho_2, FL_2 \rangle \wedge \text{env}_2 \preceq \rho_2]$

Now, we construct the required reduction sequence in two stages. In the first stage, the reductions come from  $E_2$ . In the second stage, the reductions come from  $E_1$ . This is the precise formulation of the operational interleaving alluded to in the discussion preceding the statement of this lemma.

From hypothesis,  $g_2 \preceq E_2$  and  $\langle E_2, \#, \star, \star\star \rangle \xrightarrow{*}_s \langle D_2, \#, \rho_2, FL_2 \rangle$ , there is a reduction sequence from  $\langle D_2, \#, \rho_2, FL_2 \rangle \langle D_1, \#, \rho_1, FL_1 \rangle$ , such that  $\text{env}_1 \preceq \rho_1$ .

Similarly, since  $\langle E_1, \#, \star, \star\star \rangle \xrightarrow{*}_s \langle D_1, \#, \rho_1, FL_1 \rangle$  and  $g_1 \preceq E_1$ , there is a reduction sequence from  $\langle D_1, \#, \rho_1, FL_1 \rangle$  to  $\langle D', \#, \rho_{res}, FL' \rangle$ , such that  $\text{env}_f \preceq \rho_{res}$ . ■

### B.3 Full proof

This subsection is devoted to proving that  $\mathcal{E}[[e]] \preceq e$ , for all expressions  $e$ . This section generalizes the ideas contained in the case of structural induction treated above. This section is organised as follows. First, we define the inclusive predicates for relating expressions, function symbols and operator symbols to closure operators of appropriate type. Then, we present the proof of the interleaving lemma. This proof generalizes the ideas contained in the proof of lemma ?? to arbitrary sets of semantic equations. The remainder of the proof proceeds by structural induction on the formation of expressions. The proof is a sequence of lemmas: each lemma being a straightforward reduction to the interleaving lemma. Finally, the proof is done for function definitions. The tools developed previously are used to reduce the proof to a routine fixed point induction [?].

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

**Definition 9**  $f \preceq_x e$  is defined as follows. Let

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

Then, given  $\langle \{x = e\}, \# , \star, \star\star \rangle \mapsto_s^* \langle D, \# , \rho, FL \rangle$ , where  $x$  is any variable name,

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

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

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

**Definition 10**  $f \preceq e \Leftrightarrow (\forall x) [f \preceq_x e]$  as follows.

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

The above two definitions can be combined and generalized to sets of equations generalized to a set of expressions as follows.

**Definition 11** Let  $f$  be a closure operator on  $ENV \times \underbrace{V \times V \dots}_n$ . Then,  $f \preceq_{(x_1 \dots x_n)} (E, e_1 \dots e_n)$  is defined as follows. let

- $f \langle env, \langle a_{f_1} \dots a_{f_n} \rangle \rangle = \langle env', \langle r_{f_1} \dots r_{f_n} \rangle \rangle$
- $env \preceq \rho$
- $(\forall 1 \leq i \leq n) [a_{f_i} \preceq (x_i, \rho)]$

Then, given  $\langle E \cup \{x_1 = e_1 \dots x_n = e_n\}, \#, \star, \star\star \rangle \xrightarrow{*}_s \langle D, \#, \rho, FL \rangle$ ,

$$(\forall \langle env_f, \langle r_{f_1} \dots r_{f_n} \rangle \rangle \sqsubseteq \langle env', \langle r_1 \dots r_n \rangle \rangle$$

$$(\exists) [\langle D, \#, \rho, FL \rangle \xrightarrow{*} \langle D', \#, \rho_{res}, FL' \rangle \wedge (\forall 1 \leq i \leq n) [r_{f_i} \preceq \langle \rho_{res}, x \rangle]]$$

The following lemma is the analogue of lemmas ???. The proof is immediate and is omitted.

**Lemma 9** Inclusivity and monotonicity properties of  $\preceq$ :

1.  $g \preceq E \wedge g' \sqsubseteq g \Rightarrow g' \preceq E$
2. Let  $\{g_i | i\}$  be a chain in the space of closure operators on  $ENV$ . Then,  $(\forall i) [g_i \preceq E]$  implies  $\bigsqcup_i \{g_i | i\} \preceq E$
3.  $f \preceq e \wedge f' \sqsubseteq f \Rightarrow f' \preceq e$
4. Let  $\{f_i | i\}$  be a chain in the space of closure operators on  $ENV \times V$ . Then,  $(\forall i) [f_i \preceq e]$  implies  $\bigsqcup_i \{f_i | i\} \preceq e$
5.  $f \preceq \langle e_1 \dots e_n \rangle \wedge f' \sqsubseteq f \Rightarrow f' \preceq \langle e_1 \dots e_n \rangle$
6. Let  $\{f_i | i\}$  be a chain in the space of closure operators on  $ENV \times V$ . Then,  $(\forall i) [f_i \preceq \langle e_1 \dots e_n \rangle]$  implies  $\bigsqcup_i \{f_i | i\} \preceq \langle e_1 \dots e_n \rangle$

Recall that the denotation of the function symbol was a closure operator on  $V \times V$ . The following definition relates closure operators on  $V \times V$  to the function symbol.

**Definition 12**  $s \preceq_{(x_1, x_2)} F$  is defined as follows. Let

- $f \langle a_1, a_2 \rangle = \langle b_1, b_2 \rangle$
- $a_1 \preceq (x_1, \rho) \wedge a_2 \preceq (x_1, \rho)$

Then, given  $\langle \{x_2 = F(x_1)\}, \#, \star, \star\star \rangle \mapsto_s^* \langle D, \#, \rho, FL \rangle$

$(\forall \langle b_{1f}, b_{2f} \rangle \sqsubseteq \langle b_1, b_2 \rangle) (\exists) [\langle D, \#, \rho, FL \rangle \xrightarrow{*} \langle D', \#, \rho_{res}, FL' \rangle \wedge b_{2f} \preceq \langle \rho_{res}, x_2 \rangle \wedge b_{1f} \preceq \langle \rho_{res}, x_1 \rangle]$

As before, we define  $s \preceq F$  by quantifying over all variables.

**Definition 13**  $s \preceq F \Leftrightarrow (\forall x_1, x_2) [s \preceq_{(x_1, x_2)} F]$

Recall that the denotation of constant function symbols  $op$  of arity  $n \geq 0$  is a closure operator on  $\underbrace{V \times V \times \dots}_{n+1}$ . The following definition relates closure operators  $t$  on  $\underbrace{V \times V \times \dots}_{n+1}$  to the symbol  $op$ .

**Definition 14**  $t \preceq_{(y, x_1, \dots, x_{n+1})} op$ . Let

- $t\vec{a} = \vec{b}$
- $a_1 \preceq (y, \rho) \wedge (\forall 1 \leq i \leq n) [a_{i+1} \preceq (x_i, \rho)]$

Then, given  $\langle \{y = op(x_1, \dots, x_n)\}, \#, \star, \star\star \rangle \mapsto_s^* \langle D, \#, \rho, FL \rangle$

$(\forall \vec{b}_f \sqsubseteq \vec{b} (\exists) [\langle D, \#, \rho, FL \rangle \xrightarrow{*} \langle D', \#, \rho_{res}, FL' \rangle \wedge b_{1f} \preceq \langle \rho_{res}, y \rangle \wedge (\forall 2 \leq i \leq n+1) [b_{(i+1)f} \preceq \langle \rho_{res}, x_i \rangle]$

As before, we define  $t \preceq op$  by quantifying over all variables.

**Definition 15**  $s \preceq op \Leftrightarrow (\forall x_1 \dots x_n) [t \preceq_{(x_1, \dots, x_n)} op]$

Below, we abstract out the essential property of the operators that we need for the adequacy proof.

**Definition 16** Let  $op$  be an  $n$ -ary operator,  $n$  a positive integer.  $op$  is safe, if for any configuration  $\langle D, e, \rho, FL \rangle$ :  $(\forall 1 \leq i \leq n) [v_i \preceq (x_i, \rho)] \Rightarrow [op(v_1, \dots, v_n) \preceq (op(x_1 \dots x_n), \rho)]$

Thus, the operator and the order structure interact smoothly. This property holds for the basic arithmetic operations and binary array operations, such as  $e_1[e_2]$ , and the array creating operation  $Array()$ . We assume that all the operators that we use are *safe* in this sense. The main consequence of *safety* is that  $\mathcal{E}[op] \preceq op$ .

### Interleaving lemma

Recall that all the cases of the semantics were described in terms of solving sets of equations. The following lemma works for all the types of equations that are used in the semantics. In the following lemma, we consider a general set of equations encompassing any number of definitions, function applications, operators of any arity.

**Lemma 10** Let

- $g_1, \dots, g_m, 0 \leq m$ , be closure operators on  $ENV$ , such that  $g_1 \preceq E_1 \dots g_m \preceq E_m$ .
- Let  $f_1, \dots, f_n, 0 \leq n$ , be closure operators on  $ENV \times V$ , such that  $f_i \preceq_{x_i} e_i$ .
- Let  $s_1, \dots, s_p, 0 \leq p$ , be closure operators on  $V \times V$ , such that  $F \preceq_{(x_{n+2*i-1}, x_{n+2*i})}$ .
- Let  $t_1^2 \dots t_{r_2}^2, 0 \leq r_2$ , be closure operators on  $V \times V \times V$  such that  $t_i^2 \preceq_{(x_{(n+2*p+3*i-2)}, \dots, x_{(n+2*p+3*i)})} op^2$ , where  $op^2$  is some binary operator.
- Let  $t_1^1 \dots t_{r_1}^1, 0 \leq r_1$ , be closure operators on  $V \times V \times V$  such that  $t_i^1 \preceq_{(x_{(n+2*p+3*r_2+2*i-1)}, x_{(n+2*p+3*r_2+2*i)})} op^1$  where  $op^1$  is some unary operator.

Let  $h$  be the closure operator on  $ENV \times \underbrace{V \times V \dots}_{n+2*p+3*r_2+2*r_1}$  defined as follows:

$$h\langle env, \vec{a} \rangle = \mathbf{lcs} \left\{ \begin{array}{l} env \sqsubseteq env' \\ \vec{a} \sqsubseteq \vec{a}' \\ env' = g_i env', i = 1 \dots m \\ \langle env', a'_i \rangle = f_i \langle env', a'_i \rangle, i = 1 \dots n \\ \langle a'_i, a'_{i+1} \rangle = s_k \langle a'_i, a'_{i+1} \rangle, i = (n+1) \dots (n+2*p-1) \\ \langle a'_i, a'_{i+1}, a'_{i+2} \rangle = t_{(i-n+2*p)} \langle a'_i, a'_{i+1}, a'_{i+2} \rangle, i - n - 2*p = 1 \dots (3*r_2 - 2) \\ \langle a'_i, a'_{i+1} \rangle = t_{(i-n-2*p)} \langle a'_i, a'_{i+1} \rangle, i - n - 2*p - 3*r_2 = 1 \dots (2*r_1 - 1) \\ \text{Semeq} \end{array} \right.$$

$\mathbf{in} \langle env', \langle a'_1 \dots a'_{n+2*p+3*r_2+2*r_1} \rangle \rangle$

where *Semeq* is a set of equalities of the form  $w_1 = w_2$ , where  $w_1, w_2 \in \{a'_1 \dots a'_{n+2*p+3*r_2+2*r_1}\}$ . Then,

$$\langle \cup_i E_i \cup Syneq, e_1 \dots e_n, \underbrace{F, F \dots}_p, \underbrace{op^2, \dots, op^2}_{r_2}, \underbrace{op^1, \dots, op^1}_{r_1} \rangle \preceq_{\vec{x}} h$$

where, *Syneq* is a list of equations of form determined by *Semeq*, with a equation  $\{x_i = x_j\}$  for every  $a'_i = a'_j$  in *Semeq*.

*Proof:* Rather than doing the proof for the general case that involves multiple indices, we sketch the special case of the proof that resembles the semantic definition for an expression of form  $e_1 op e_2$ . The sketch is in sufficient detail to bring out the resemblance to the proof of lemma ???. In particular,  $m = 0, n = 2, p =$

$0, r_1 = 0, r_2 = 1$ . The definition of  $h$  now takes the form:

$$h\langle env, \vec{a} \rangle = \mathbf{lcs} \left\{ \begin{array}{l} \vec{a} \sqsubseteq \vec{b} \\ env \sqsubseteq env' \\ \langle env', b_1 \rangle = f_1 \text{pairenv}' b_1 \\ \langle env', b_2 \rangle = f_2 \text{pairenv}' b_2 \\ \langle b_3, b_4, b_5 \rangle = t\langle b_3, b_4, b_5 \rangle \\ b_1 = b_4 \\ b_2 = b_5 \end{array} \right. \\ \mathbf{in} \langle env', \vec{b} \rangle$$

First, we alter the definitions of the given closure operators so that their types match the type of  $h$ . Thus, the domain of definition of each closure operator is made to be  $ENV \times \underbrace{V \times V \dots}_5$ . This is done in the natural way. The new closure operators only change some of the  $V$  components and leave the others untouched. It will turn out that  $h$  can be recovered as the least upper bound of the closure operators so defined. More precisely, define closure operators  $h'_1, \dots, h'_4$ , as follows:

- Let  $f_1\langle env, a_1 \rangle = \langle env', b_1 \rangle$ . Define  $h'_1\langle env, \vec{a} \rangle = \langle env', \langle b_1, a_2 \dots a_5 \rangle \rangle$ .
- Let  $f_2\langle env, a_2 \rangle = \langle env', b_2 \rangle$ . Define  $h'_2\langle env, \vec{a} \rangle = \langle env', \langle a_1, b_2, a_3 \dots a_5 \rangle \rangle$ .
- Let  $t\langle a_3, a_4, a_5 \rangle = \langle b_3, b_4, b_5 \rangle$ . Define,  $h'_3\langle env, \vec{a} \rangle = \langle env, \langle a_1, a_2, b_3, b_4, b_5 \rangle \rangle$
- $h'_4\langle env, \vec{a} \rangle = \langle env, \vec{b} \rangle$ , where  $b_1 = b_4 = a_1 \sqcup a_4$  and  $b_2 = b_5 = a_2 \sqcup a_5$  and  $b_3 = a_3$ .

Let  $u = (h'_1 \circ \dots \circ h'_4)$ . Note that  $h = \bigsqcup_l \{u^l \mid l = 1, 2, \dots\}$ . So, from lemma ??, it suffices to prove  $(\forall l) [u^l \preceq_{\vec{x}} \text{conf}]$ , where  $\text{conf}$  is  $\{x_1 = x_4, x_2 = x_5, x_1 = e_1, x_2 = e_2, x_3 = \text{op}(x_1, x_2)\}$ .

This is done by induction on  $l$ . This proof follows closely the proof of lemma ?? and is omitted. ■

## Cases of Structural Induction

The following lemma proves a number of cases of structural induction by reduction to the interleaving lemma.

**Lemma 11** (*Cases of structural induction*):

**Definitions:**  $f \preceq e \Rightarrow g \preceq x = e$  where  $g$  is the closure operator on  $ENV$  defined as

$$g \text{ env} = \mathbf{lcs} \left\{ \begin{array}{l} env \sqsubseteq env' \\ \langle env', b \rangle = f\langle env', b \rangle \\ b = env'[x] \end{array} \right. \\ \mathbf{in} \langle env', b \rangle$$

**Expressions in contexts:**  $[f \preceq e \wedge g \preceq defs] \Rightarrow h \preceq (defs \text{ in } e)$ , where  $h$  is the closure operator on  $ENV \times V$  defined as

$$h \langle env, a \rangle = \begin{array}{l} \mathbf{lcs} \left\{ \begin{array}{l} \langle env, a \rangle \sqsubseteq \langle env', b \rangle \\ \langle env', b \rangle = f \langle env', b \rangle \\ env' = g \ env' \end{array} \right. \\ \mathbf{in} \langle env', b \rangle \end{array}$$

**Binary operators:**  $f_1 \preceq e_1 \wedge g \preceq e_2 \Rightarrow h \preceq e_1 \text{ op } e_2$ , where  $op$  is safe and  $h$  is the closure operator on  $ENV \times V$  defined as

$$h \langle env, a \rangle = \mathbf{lcs} \left\{ \begin{array}{l} \langle env, a \rangle \sqsubseteq \langle env', b \rangle \\ \langle env', b_1 \rangle = f_1 \langle env', b_1 \rangle \\ \langle env', b_2 \rangle = f_2 \langle env', b_2 \rangle \\ b_1 \text{ op } b_2 \sqsubseteq b \end{array} \right. \\ \mathbf{in} \langle env', b \rangle$$

**Array creation:**  $f \preceq e \Rightarrow g \preceq \text{array}(e)$ , where  $g$  is a closure operator on  $ENV \times V$  defined as

$$g \langle env, a \rangle = \mathbf{lcs} \left\{ \begin{array}{l} env \sqsubseteq env' \\ env', n = f_1 \langle env', n \rangle \\ r = \text{Array}(n) \sqcup a \end{array} \right. \\ \mathbf{in} \langle env', r \rangle$$

**Function Application:**  $F \preceq s \wedge f \preceq e \Rightarrow f \preceq F(e)$ , where  $h$  is a closure operator on  $ENV \times V$  defined as

$$h \langle env, a \rangle = \mathbf{lcs} \left\{ \begin{array}{l} env \sqsubseteq env' \\ \langle env', v \rangle = f \langle env', v \rangle \\ \langle v, r \rangle = s \langle v, r \rangle \end{array} \right. \\ \mathbf{in} \langle env', r \rangle$$

*Proof:* The proofs are a straightforward application of the interleaving lemma ??.

1. Consider  $g'$ , a closure operator on  $ENV \times V \times V$  defined as:

$$g' \langle env, \langle a_1, a_2 \rangle \rangle = \mathbf{lcs} \left\{ \begin{array}{l} env \sqsubseteq env' \\ \vec{a} \sqsubseteq \vec{b} \langle env', b_1 \rangle = f \langle env', b_1 \rangle \\ b_2 = env'[x] \\ b_1 = b_2 \end{array} \right. \\ \mathbf{in} \langle env', b \rangle$$

From assumption  $f \preceq e$ ,  $f \preceq_x e$ , for all  $x$ . From lemma ??,  $g' \preceq_{(x,x)} \langle x = x, e \rangle$ , for all  $y$ . Also, note that  $g' \langle env, \langle \perp, \perp \rangle \rangle = \langle env', \langle \perp, \perp \rangle \rangle$ , where  $g \ env = env'$ . Thus,  $g \preceq x = e$ .

2. Immediate from lemma ??.

3. Consider  $h'$  the closure operator on  $ENV \times \underbrace{V \times \dots V}_5$ , defined as,

$$h' \langle env, \vec{a} \rangle = \mathbf{lcs} \left\{ \begin{array}{l} \langle env, \vec{a} \rangle \sqsubseteq \langle env', \vec{b} \rangle \\ \langle env', b_1 \rangle = f_1 \langle env', b_1 \rangle \\ \langle env', b_2 \rangle = f_2 \langle env', b_2 \rangle \\ b_3 \text{ op } b_4 \sqsubseteq b_5 \\ b_1 = b_3 \\ b_2 = b_4 \end{array} \right. \mathbf{in} \langle env', \vec{b} \rangle$$

From assumptions  $f_1 \preceq e_1$ ,  $f_2 \preceq e_2$  and lemma ??,

$$(\forall x_1, x_2, x_3) [h' \preceq x_1, x_2, x_1, x_2, x_3 \langle e_1, e_2, x_3 = op(x_1, x_2) \rangle]$$

Let  $x_3$  be any variable name, and

- $f \langle env, a_f \rangle = r$
- $env \preceq \rho$
- $a_f \preceq (x, \rho)$
- $\langle \{x = e\}, \#, \star, \star\star \rangle \xrightarrow{*}_s \langle D, \#, \rho, FL \rangle$

$\langle \{x_3 = e_1 \text{ op } e_2\}, \#, \star, \star\star \rangle \longrightarrow \langle \{x_3 = x_1 \text{ op } x_2, x_1 = e_1, x_2 = e_2\}, \#, \star, \star\star \rangle$ , where  $x_2, x_3$  are in  $FL$ . From lemma ??, there is a configuration  $conf$  such that

$$\langle D, \#, \rho, FL \rangle \xrightarrow{*} conf \wedge \langle \{x_3 = x_1 \text{ op } x_2, x_1 = e_1, x_2 = e_2\}, \#, \star, \star\star \rangle \xrightarrow{*}_s conf$$

So, without loss of generality, we can assume that  $\langle \{x_3 = e_1 \text{ op } e_2\}, \#, \star, \star\star \rangle \xrightarrow{*}_s \langle D, \#, \rho, FL \rangle$ .

Result follows from  $h' \preceq x_1, x_2, x_1, x_2, x_3 \langle e_1, e_2, x_3 = op(x_1, x_2) \rangle$ .

4. Proof of the case of Array creating expressions follows in a similar fashion using lemma ??.

5. Proof of the case of application follows in a similar fashion using lemma ??.

The machinery developed so far renders the proofs of the remaining cases, except that of functional application, routine. These proofs are omitted.

**Lemma 12** (Other cases of structural induction):

1. **Conditional expressions:**

Let  $f_1 \preceq e_1$ ,  $f_2 \preceq e_2$  and  $f_3 \preceq e_3$ . Then,  $g \preceq \text{cond}(e_1, e_2, e_3)$  where  $g$  is the closure operator on  $ENV \times V$  defined as follows:

$$g\langle env, a \rangle = \mathbf{let} \ f_1\langle env, \perp \rangle = \langle env', bool \rangle \ \mathbf{in} \\ \mathbf{if} \ bool \ \mathbf{then} \ f_2\langle env', a \rangle \\ \mathbf{else} \ f_3\langle env', a \rangle$$

2. **Base case for arrays :**  $\mathcal{E}[[L_1 \dots L_n]] \preceq [L_1 \dots L_n]$

3. **Base case for variables :**  $\mathcal{E}[[x]] \preceq x$

**Function definition**

The proof that  $\mathcal{F}[[F]] \preceq F$  proceeds essentially by fixpoint induction. The following lemma formalizes the induction step.

**Lemma 13**  $s \preceq F \Rightarrow \tau(s) \preceq F$ , where  $\tau$  is the continuous function on closure operators on  $ENV \times V$ , defined as below.

$$\tau(f) = \lambda (v, a). \\ \mathbf{lcs} \left\{ \begin{array}{l} \{x \mapsto v, y \mapsto \perp, F \mapsto f\} \\ \sqsubseteq env' \\ a \sqsubseteq r \\ env \sqsubseteq env' \\ env' = \mathcal{C}[[def - list]] \ env' \\ \langle env', r \rangle = \mathcal{E}[[exp]] \ env' \ r \end{array} \right. \\ \mathbf{in} \ \langle env'[x], r \rangle$$

Recall that  $\mathcal{F}[[F]]$  was defined in Section ?? as the least fixpoint of  $\tau$ .

*Proof:* Proof is a simple structural induction using hypothesis  $s \preceq F$ , and the previous lemmas that built up the cases of structural induction. Note that  $\tau(s)(f)$  is the closure operator on  $ENV \times V$  defined as

$$\tau(s)(f) \ env \ a = \\ \mathbf{lcs} \left\{ \begin{array}{l} env \sqsubseteq env' \\ a \sqsubseteq r \\ \langle env', v \rangle = f\langle env', v \rangle \\ v, r = \tau(s)\langle v, r \rangle \end{array} \right. \\ \mathbf{in} \ \langle env', r \rangle$$

The above can be rewritten as

$$\tau(s)(f) \text{ env } a = \text{ics} \left\{ \begin{array}{l} \text{env} \sqsubseteq \text{env}' \\ \{x \mapsto v, y \mapsto \perp, F \mapsto s\} \sqsubseteq \text{env}'' \\ a \sqsubseteq r \\ \langle \text{env}', v \rangle = f \langle \text{env}', v \rangle \\ \text{env}'' = \mathcal{C}[\text{def} - \text{list}] \text{ env}'' \\ \langle \text{env}'', r \rangle = \mathcal{E}[\text{exp}] \text{ env}'' \ r \\ \text{env}''[x] \sqsubseteq \text{env}'[x] \end{array} \right. \\ \text{in } \langle \text{env}', r \rangle$$

From inductive hypothesis  $s \preceq F$  and using lemmas ?? ??, we have  $\mathcal{C}[\text{def} - \text{list}] \preceq \text{def} - \text{list}$  and  $\mathcal{E}[\text{exp}] \preceq \text{exp}$ . Another use of lemmas ?? ?? allows us to deduce the desired result,  $\tau(s)(f) \preceq F(e)$ . ■

**Lemma 14**  $\mathcal{F}[F] \preceq F$ .

*Proof:* It is easy to check that  $\perp \preceq F$ , where  $\perp$  is the identity function on  $V \times V$ , the least closure operator on  $V \times V$ . From lemma ??, we deduce  $(\forall k) [\tau^k(\perp) \preceq F]$ . As an analogue of lemmas ??, ?? which prove the inclusivity of various forms of the predicate  $\preceq$ , we have  $(\forall k) [\tau^k(\perp) \preceq F] \Rightarrow \mathcal{F}[F] \preceq F$ . Hence, we have  $\mathcal{F}[F] \preceq F$  ■

The main theorem is now reduced to a simple structural induction proof.

**Theorem 5** For all expressions  $e$ ,  $\mathcal{E}[e] \preceq e$

*Proof:* The base cases is proved in lemma ??. The structural induction steps follow from lemmas ??, ??. ■

## C Full-abstraction

In full-abstraction we aim to establish that the denotational semantics is an accurate guide to program behavior 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 behavior of syntactic expressions as follows.

**Definition 17**  $e_1 \sqsubseteq_{op} e_2$  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 6** *The denotational semantics is inequationally adequate i.e*

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

*Proof:* Let  $\mathcal{E}[e_1] \sqsubseteq \mathcal{E}[e_2]$  and  $\langle D[e_1], C[e_1], \Phi, FL \rangle \rightarrow b$ . Consider  $\mathcal{M}[\langle D[e_1], C[e_1], \Phi, FL \rangle] \perp \perp$ . Since one step reduction preserves meaning, we deduce that

$\mathcal{M}[\langle D[e_1], C[e_1], \Phi, FL \rangle] \perp \perp = env, b$  for some  $env$ . It can easily be shown that the context operations are monotone. Consequently, we have  $env, b \sqsubseteq \mathcal{M}[\langle D[e_2], C[e_2], \Phi, FL \rangle] \perp \perp$ .

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

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

Note that  $\langle \Phi, F(x_1, \dots, x_n), \Phi, FL \rangle \rightarrow \langle D[e_2], C[e_2], \Phi, FL \rangle$ . Since one-step reduction preserves meaning,

$$\mathcal{E}[F(x_1, \dots, x_n)] \perp \perp = \mathcal{M}[\langle D[e_2], C[e_2], \Phi, FL \rangle] \perp \perp$$

Hence,  $\perp, b \sqsubseteq \mathcal{E}[F(x_1, \dots, x_n)] \perp \perp$ . So we have one of the following.

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

From 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 prime elements of the space of the closure operators on  $ENV \times V$  are expressible as the meanings of expressions. As in Plotkin's proof of full-abstraction for PCF [?], the crux of the proof below is the construction of contexts that can semantically distinguish two different expressions.

**Theorem 7** *The denotational semantics is fully-abstract i.e.*

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

The forward implication was proved in the previous theorem. For the reverse implication consider the case when  $\mathcal{C}[[e_1]] \not\sqsubseteq \mathcal{C}[[e_2]]$ . Since the semantic domains are algebraic

$$\mathcal{C}[[e_1]] \not\sqsubseteq \mathcal{C}[[e_2]] \implies$$

$$(\exists \langle env_1, v_1 \rangle, \langle env_2, v_2 \rangle)$$

$$[f_{\langle env_1, v_1 \rangle \Rightarrow \langle env_2, v_2 \rangle} \sqsubseteq \mathcal{E}[[e_1]] \wedge f_{\langle env_1, v_2 \rangle \Rightarrow \langle env_2, v_2 \rangle} \not\sqsubseteq \mathcal{E}[[e_2]]]$$

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

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

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.

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

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. Note that

$$\begin{aligned} \mathcal{E}[[F(e_1)]] \perp \perp &= \mathcal{M}[[\langle \Phi, F_{e_1}, \Phi, FL \rangle]] \perp \perp \\ &= \mathcal{M}[[\langle G, \text{if } E' \text{ then } 0 \text{ else } 1, \phi, FL \rangle]] \perp \perp \end{aligned}$$

where  $G = E \cup E' \cup \{x = e_1\}$ . However, we have

$$\mathcal{E}[\langle G, \text{if } E' \text{ then } 0 \text{ else } 1, \phi, FL \rangle] \perp \perp =$$

$$\text{lcs} \left\{ \begin{array}{ll} \mathcal{C}[E] \text{ env} = \text{env} & 1 \\ \mathcal{C}[x = e_1] \text{ env} = \text{env} & 2 \\ \mathcal{C}[E'] \text{ env} = \text{env} & 3 \\ \mathcal{E}[C[x]] \text{ env } a = a & 4 \end{array} \right.$$

$$\text{in } \langle \text{env}, a \rangle$$

Equation 1 merely asserts that  $\text{env}_1 \sqsubseteq \text{env}$ . Equation 3 ensures that  $v_1 \sqsubseteq \text{env}[x]$ . Now equation 2 ensures that  $\text{env}_2 \sqsubseteq \text{env}$  and  $\text{env}[x] \sqsubseteq v_2$ . So, we deduce that  $\text{env}_2, 0 \sqsubseteq \mathcal{E}[F(e_1)] \perp \perp$ . Also,  $\langle \text{env}_2, 0 \rangle$  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*. Hence,  $\mathcal{E}[F(e_1)] \perp \perp = T$ . That implies that the least common solution of equations 1, 2 and 3 is  $T$ . Hence, we deduce that  $\mathcal{E}[e_2] \text{ env}_1 \ v_1 = T$ , which contradicts the fact that  $f_{\langle \text{env}_1, v_1 \rangle \Rightarrow \langle \text{env}_2, v_2 \rangle} \not\sqsubseteq \mathcal{E}[e_2]$ . 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. Hence  $\perp, 0 \sqsubseteq \mathcal{E}[F(e_1)] \perp \perp$ . That implies that the least common solution of equations 1, 2 and 3 is greater than  $\langle \text{env}_2, v_2 \rangle$ . Hence, we deduce that  $\text{env}_2, v_2 \sqsubseteq \mathcal{E}[e_2] \text{ env}_1 \ v_1$ . This contradicts  $f_{\langle \text{env}_1, v_1 \rangle \Rightarrow \langle \text{env}_2, v_2 \rangle} \not\sqsubseteq \mathcal{E}[e_2]$ . Hence,  $\langle \Phi, F(e_2), \Phi, FL \rangle$  does not reduce to 0.

This completes the proof of full-abstraction for Cid. ■