Closure Operator Semantics for Concurrent Constraint Logic Programming

Radha Jagadeesan*
Vasant Shanbhogue

TR 90-1174
December 1990

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

*This research was supported partly by the National Science Foundation under Grant No. CCR-8818979 and partly by a fellowship from the Mathematical Sciences Institute, Cornell.

Closure Operator semantics for concurrent constraint logic programming

Radha Jagadeesan *
Dept.of Computer Science
Cornell University

Vasant Shanbhogue
Dept. of Computer Science
Wichita State University

Abstract

This paper develops a denotational and abstract model based on closure operators for concurrent constraint logic programming. The denotational semantics is built domain theoretically and not from the computation sequences. The denotational semantics is related to an operational semantics. The operational semantics distinguishes successful and unsuccessful computations and observes intermediate results of divergent computations. The paper extends to the indeterminate setting, previous work on functional languages with logic variables [9].

1 Introduction

A fundamental problem in the theory of concurrency is integrating concurrency with abstraction. Kahn's work on determinate dataflow [10] is an example where concurrency meshes smoothly with abstraction. In the denotational model, the internal operational details are abstracted away and processes are viewed as continuous functions on streams. The match between the two different views of processes is quite tight. The denotational semantics is an accurate guide to operational behaviour in all contexts. Kahn's semantics was restricted to determinate behaviours with unidirectional flow of information. This has motivated the study of more general models of computation, for example, process algebras [14, 6].

Logic programming generalises Kahn's original model in two ways. The presence of OR-parallelism entails indeterminacy. The bidirectional nature of the flow of information comes from unification. Pure logic programming has an elegant declarative basis based on viewing computations as proofs carried out with a single inference rule, namely resolution. However, efficiency considerations have motivated the addition of various control features. These control constructs enable the use of logic programming for specification and implementation of systems of processes. In the presence of these control constructs, the correspondence of computations of logic programs to proofs in first order logic becomes weak and tenuous. This is the motivation for investigations into an abstract semantics for logic programs. Motivated by Kahn's elegant model of static dataflow, it is natural to demand that the semantics provide the missing declarative framework for concurrent logic programs. Thus, the denotational semantics should provide a conceptually simpler view of processes than the operational semantics.

*This research was supported partly by the National Sciences Foundation under Grant No. CCR-8818979 and partly by a fellowship from the Mathematical Sciences Institute, Cornell

This paper is intended to be a piece in this program. The main contribution of the paper is a fully-abstract semantics of concurrent constraint programming languages similar to Flat GHC [22]. The language has input matching and committed choice (also called don't know nondeterminism) and flat guard predicates. Related and more powerful languages have been studied extensively [22] [18] [26].

The main feature of the semantics is that it captures enough information about processes to make purely local reasoning about processes sufficient for reasoning in arbitrary contexts. This is achieved without resorting to operational notions like renaming. Our semantics follows previous work in the semantics of functional languages with logic variables [9]. This work views unification as constraint-solving and identifies closure operators [21] as the domain theoretic models of constraints. The semantics is couched in terms of solving of equations. The semantics in this paper can be viewed as the extension of these ideas to an indeterminate setting. Our constructions can be interpreted as the domain-theoretic analogues of ideas in previous work on semantics of logic programming without occur check [27] [12].

Two key points are worth mentioning. Firstly, the use of closure operators as the meanings of processes makes explicit the bidirectional flow of information. Secondly, the mathematical structures that we use in the denotational semantics are naturally endowed with a notion of parallel composition. There is an elegant characterisation of the parallel composition operation as set intersection. The semantics makes no atomicity assumptions and handles infinite behaviours. Furthermore, the semantics works correctly only under the assumption of AND-fairness. The structure used to model indeterminacy is a variant [8] of the standard powerdomains used to model indeterminacy in imperative languages [15, 23].

It is worthwhile to contrast our techniques against the existing work. There is a denotational model for a more powerful language than the one we are considering [11]. The semantics is based on a powerdomain of resumptions, and can be viewed as explicitly keeping track of the flow of data and demand tokens. The relationship between the two models is not addressed in this work. A full abstraction result has been proved for a related and more powerful language [17]. The semantics gives the meaning of processes as sets of pairs of substitutions and suspensions. The meaning of parallel composition is got by explicitly combining these sets. The most significant difference between our approaches is in the observations allowed. In keeping with the constraint view, we think of failed computations as imposing inconsistent constraints on the environment and do not observe the substitutions of failed computation. There is a fully abstract semantics for a language based on the Ask-Tell paradigm [26]. The semantics is inspired by work in process theory, and intuitively associates processes with the set of the sequence of interactions on possible paths to a point of data quiescence. Both the above papers [17] [26] do not handle infinite computations.

The significant difference between this work and the papers cited above arises from the different motivations. Previous work [17] [26] can be viewed as starting out with some reasonable notion of observation and attempting to find the minimal information that needs to be encoded to be able to distinguish programs. Rather informally, this extra information took the form of *interactions with the environment along every possible computation path.* The aim of this paper is to simplify the programmer's view of processes and is intended as a programmer's first approximation in thinking about processes. In fact, aided by powerful tools from domain theory, the semantics here essentially presents a process as an input-output relation: i.e the possible output environments got by executing a process in a given input environment. However, this enormous simplification is attained at some cost: the semantics identifies programs that are distinguishable in previous work.

In particular, the semantics is inadequate for studying "progress properties". The full-abstraction result identifies *precisely* the notion of distinguishability that the semantics captures. We hope to convince the reader that the view of observability modelled by the semantics is a significant and non-trivial subset of the observations modelled in previous work. Thus, we hope to convince the reader that the work here constitutes significant progress towards a declarative semantics for concurrent constraint logic programming.

The paper is organised as follows. First, we sketch the operational semantics of a less powerful language. This language is intended to make the exposition clearer, and establish the connections with datflow that help to understand the semantics. Next, we give a detailed description of our notion of tests. In the next section, we decribe the domain theoretic structures needed for the denotational model. Then, we describe the denotational semantics. In the next section, we sketch the proof of full-abstraction. The detailed proofs are rather long and are presented in the appendices. In the final section, we sketch the extra strutures needed to model the full language.

2 Transition system

In this section, we define the language studied first. The language is very similar to the language flat GHC [22]. This is a restriction of the more general language GHC [25], to have only flat guard predicates. The language that we study initially is more restrictive. The basic difference is the restrictions on the predicates in the guards. Intuitively, the restrictions on the guard predicates amount to restricting the guard predicates to look at only the values of variables. For example, we allow checks of the form $x = c$, where c is a 0-ary fucntion symbol (constant). However, checks of the type $x = y$, for x and y variable names are disallowed. Even with this restriction, it is possible to code typical programs, such as the short-circuit protocol. Furthermore, we wish to emphasize that this restriction is only for expository purposes. The intention is to motivate the dataflow view of computation. These restrictions are removed in the last section of this paper.

We follow standard syntax,as for example in [22].

Definition 1 *(Definition of some standard syntactic entities)*

- $x, L \epsilon$ *Id = countable set of identifiers*
- $f \epsilon$ *Functions = set of function symbols*
- $p \epsilon$ *Pred = set of predicate symbols*
- $t \epsilon$ *Terms = Variables or a function symbol of arity n applied to n terms.*
- $A, B \epsilon$ *Atoms* $= p(t_1, \ldots t_n)$*, where p is an n ary predicate and t_i are terms.*

For presenting the operational semantics, we need a notion of environment. The following definitions follow earlier work [9].

$E \epsilon$ Alias-set $::= \{t_1, ..., t_n\}$

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

We use the unification algorithm, without occurs check. This has been extensively studied, for example in [20] [1] [7] [27]. This is an algorithm for the unification problem in the domain of regular infinite trees. Hence, infinite data structures are considered to be legitimate objects of computation. Below, we present the main results without proofs. 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 enviroment ρ, and A1 and A2 have an identifier in common, then $\rho \rightsquigarrow (\rho - \{A1\} - \{A2\}) \cup \{A1 \cup A2\}$.*
2. *If $\{f(t_1, ..., t_n), f(t'_1, ..., t'_n)\} \subseteq A\epsilon\rho$ then $\rho \rightsquigarrow \rho \cup \{\{t_1, t'_1\}, ..., \{t_n, t'_n\}\}$.*
3. *If $\{f(t_1, ..., t_n), g(t'_1, ..., t'_n)\} \subseteq A\epsilon\rho$ and $f \neq g$ then $\rho \rightsquigarrow error$.*

Intuitively, these 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 terms with same function symbol are in an alias-set, their arguments must be in alias-sets as well. The third clause detects constraints that are impossible to satisfy.

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

Lemma 1 *The relation $\overset{*}{\rightsquigarrow}$ has the following properties:*

1. *If $\rho_1 \overset{*}{\rightsquigarrow} \rho_2$ and $\rho_1 \overset{*}{\rightsquigarrow} \rho_3$ then $\rho_2 \overset{*}{\rightsquigarrow} \rho_4$ and $\rho_3 \overset{*}{\rightsquigarrow} \rho_4$ for some ρ_4.*
2. *There is no infinite sequence of distinct enviroments ρ_i such that $\rho_i \rightsquigarrow \rho_{i+1}$ for all i.*
3. *For any environment ρ, there is a unique, irreducible ρ_1 such that $\rho \overset{*}{\rightsquigarrow} \rho_1$.*

The first property states that reduction of environments has the Church-Rosser property. The second property states that an environment cannot be reduced indefinitely. The third property is an immediate consequence of the first two. In the rest of the paper, we will nusually not be concerned with the explicit details of the algorithm. in particular, we will usually not distinguish between syntactic environments and their reduced forms.

The presentation of the reduction relation follows [22] closely. The transition system is defined in terms of a input match and try function. A guarded clause is of form, $A \leftarrow G|B_1, \ldots B_n$, where A, B_i are atoms, and G is called the guard predicate. A concurrent constraint logic program is a set of guarded clauses. We assume that the heads A is made up of *distinct* variables. These restrictions ensure that input match never suspends or fails.

First, we define the notion of input matching. This is formalised as a function *match* that takes two terms and an environment as arguments. The definition here is more general than is needed. The exact type is given by $match : Terms \times Terms \times Environment \rightarrow \{Environments, fail\}$. This is done by structural induction on terms.

$$match(x, t, \rho) = \begin{cases} fail, \text{ if } \rho \bigcup \{x, t\} = error \\ \rho \bigcup \{x, t\}, \ otherwise \end{cases}$$

$$match(\vec{x}, \vec{u}) = \begin{cases} fail, \text{ if } \rho \bigcup \{\vec{x}), \vec{u}\} = error \\ \rho \bigcup \{\vec{x}, \vec{u}\}, \ otherwise \end{cases}$$

Match can be easily extended to a function with domain $Atoms \times Atoms \times Environment$ and range $\{Environments, fail, susp\}$

First, we define formally the value of a term in a syntactic environment. Let ρ be a syntactic environment, in reduced form, that is consistent. Consider the following transition system. Let s denote a finite sequence. Let $t = f_i^n(t_1, \ldots t_n)$ be any term. Then, we define $t \uparrow s$ inductively as follows:

- $t \uparrow 0 = f_i^n$
- $t_i \uparrow s = \mathrm{g} \Rightarrow t \uparrow [i|s] = \mathrm{g}$

The following rules "evaluate" an expression of form $t \uparrow s$ in an environment ρ.

1. $< x, \rho > \rightarrow undefined$
   if the alias set of x contains no non-variable terms.

2. $< x, \rho > \rightarrow t$
   if t is in the alias set of x. Note that there may be may be many different terms in the alias set of x. t is arbitrarily chosen from this alias set, by some rule, say lexicographic ordering. (The following lemma essentially states that this seemingly arbitrary choice does not affect the results of the evaluation of $\langle e, \rho \rangle$, in the interesting cases)

3. $\dfrac{< e, \rho > \rightarrow t}{< e \uparrow s, \rho > \rightarrow t \uparrow s}$

Lemma 2 *Let ρ be a consistent syntactic environment. Let $< e, \rho > \rightarrow f_n^i$, where f_n^i is a function symbol. Then, $< e, \rho > \rightarrow f_n^i$ is independent of the choice made in rule 2 of the transition system above.*

The guard predicate is conjunction of primitive guards. The primitive guards are one of the following:

1. $x \uparrow s = f$, $x \uparrow s \neq f$, where f is a function symbol and x is a variable name.

2. Various numeric predicates, for example $=:=, \leq, \neq:\neq,$

The meaning of a guard is defined relative to a syntactic environment ρ. We define below the meaning of one of the primitive guards. The meanings of the other primitive guards are defined similarly.

$$(x \uparrow s = f, \rho) \quad = \quad \begin{cases} fail, \text{ } if \ x \uparrow s \neq f \\ true, \text{ } if \ x \uparrow s = f \\ susp \ otherwise \end{cases}$$

The meaning of conjunction of primitive guards in a syntactic environment is defined as follows.

$$(G_1 \wedge G_2 \wedge G_n, \rho) \quad = \quad \begin{cases} true, \text{ } if (\forall i) \, [(G_i, n) = true] \\ fail, \text{ } if \ (\exists i) \, [(G_i, n) = fail] \\ susp \ otherwise \end{cases}$$

We can now define the clause try function.

$$try(t_1 = t_2, X_1 = X_2, \rho) = \begin{cases} fail, \text{ if } \rho \bigcup \{t_1, t_2\} = error \\ \rho \bigcup \{t_1, t_2\}, otherwise \end{cases}$$

$$try(A, A' \leftarrow G|B, \rho) = \begin{cases} fail, \text{ if} match(A, A', rho) = fail \\ \quad \vee[match(A, A', \rho) = \rho' \wedge \ (G, \rho') = fail] \\ \rho', \text{ if } match(A, A', \rho) = \rho' \wedge \ (G, \rho') = true \\ susp, \ otherwise \end{cases}$$

The following definitions are in the context of a concurrent constraint logic program P. The operational semantics is presented in the form of configurations. A configuration is either a pair $\langle \{C_i\}, \rho \rangle$, or of the form $fail$. In the first case, the first component of the pair is a multiset of atoms. The reduction rules are presented following the presentation of [22].

1. Reduce:
   $\langle \{A_1 \dots A_i, \dots A_n\}, \rho \rangle \longrightarrow \langle \{A_1 \dots B_1, \dots B_m, \dots A_n\}, \rho' \rangle$
   if $try(A_i, C, \rho) = \rho'$, where $C = A \leftarrow G|B_1 \dots B_n$ is some renamed apart clause of the program P.

2. Fail:
   $\langle \{A_1 \dots A_i, \dots A_n\}, \rho \rangle \longrightarrow fail$
   if for some i and for every renamed apart clause $A \leftarrow B_1 \dots B_n$ of P, $try(A_i, C) = fail$.

The supend result of the *try* function is not used in the transitions. If A is a goal atom, for which $try(A, C) = susp$ for some clause C in P, and $try(A, C') = susp \vee try(A, C') = fail$, for all clauses C' in P, A is suspended. A configuration in which all atoms are suspended is said to be deadlocked. We make a fairness assumption on the transition system. This is called **AND-fairness**. We follow the definitions of previous work [22]. Define a computation c to be a sequence of configurations $\langle S_i, \rho_i \rangle$, such that $\langle S_i, \rho_i \rangle \longrightarrow \langle S_{i+1}, \rho_{i+1} \rangle$. Then, c is said to be AND-fair, if there is no reduce or fail transition that remains enabled in almost all the configurations of c.

3 Observations and Tests

The aim of this section is to develop a theory of observations and tests, for the restricted language. The essentials of the theory go through for the full language. The minor changes that are required are discussed in section 8.

We first define the notion of finite observations. The intuitive meaning of saying that an observation is finite is that it can be made in a finite amount of time. For example, an observation of form " x = 1", is deemed to be finite. The formal statement of this idea of finiteness is in terms of the recursive enumerability of the set of observables. The set of observables that we present here forms a recursive enumerable set. The theory is closely related to previous work in the context of process calculi [5]. A notion of tests is then defined. The tests that we allow correspond roughly to placing the process in arbitrary contexts. The framework of observations and tests presented here handles infinite computations.

Recall that the syntactic environment ρ, was presented as a set of alias sets. Intuitively, finite observations of environments correspond to looking at the tree structure of finitely many variables to a finite depth.

Define a set of finite terms *Finterms* as follows:

Definition 3 *The set Finterms is defined inductively as follows:*

1. $\Omega \in Finterms$

2. $t_1 \ldots t_n \in S \Rightarrow f(t_1 \ldots f_n) \in Finterms$, *where f is an n-ary function symbol.*

Note that the set of alias sets implicitly contains the notion of a variable evaluating to a value better than a given finite term. $t \sqsubseteq \rho(x)$ if $\rho \bigcup \{x,t\} = \rho$. Note that there is a need for an extra rule for reduction in the alias sets, as we have a new symbol Ω. This rule is motivated by thinking of Ω as a symbol of no information.

$$\rho \bigcup \{A_1 \bigcup \{\Omega\}\} \rightsquigarrow \rho \bigcup \{A_1\}$$

An example will help make the idea clearer. Let $\rho = \{\{x, f(y))\}, \{y, g(z)\}\}$. Then,

$$f(\Omega) \sqsubseteq \rho(x), f(g(\Omega)) \sqsubseteq \rho(x), f(g(f(\Omega)) \not\sqsubseteq \rho(x)$$

Now, we define the notion of observations. Denote by *Primobs*, all expressions of the form $t \sqsubseteq \rho(x)$, where $t \in Finterms$. The relation $\models \subseteq SYNENV \times Primobs$ defined below models the primitive propostions true in a given syntactic environment. $\models$ is written infix.

Definition 4 *The $\models$ relation is defined as follows:*

1. $\rho \models p$, *for all $p \in Primobs$, if ρ is inconsistent.*

2. $t \sqsubseteq \rho(x) \Rightarrow \rho \models t \sqsubseteq \rho(x)$

Note the handling of inconsistent environments. The inconsistent environment satisfies any constraint. This is motivated by the constraint view of the environment [19]. An environment that has inconsistent constraints imposed on it logically implies any constraint.

Let $Term = \{success, donotcare\}$. Then, define OBS as the the set of all expressions generated from $Term \times Primobs$ using the boolean connectives $\wedge$ and $\vee$. $\vdash \subseteq CONF \times OBS$, is the relation that indicates properties true of a configuration. We assume that the variables occurring in the OBS part occur in the $CONF$ part. Let $\langle C, \rho \rangle$ be an operational configuration.

Definition 5 *(Definition of $\vdash$)*

1. $C = \langle true, \rho \rangle \wedge \rho \models p \Rightarrow \langle C, \rho \rangle \vdash \langle success, p \rangle$

2. $\rho \models p \Rightarrow \langle C, \rho \rangle \vdash \langle donotcare, p \rangle$

3. $C = fail \Rightarrow \langle C, \rho \rangle \vdash \langle donotcare, p \rangle$

4. $\langle C, \rho \rangle \vdash p_1$ *or* $\langle C, \rho \rangle \vdash p_2 \Rightarrow \langle C, \rho \rangle \vdash p_1 \vee p_2$

5. $\langle C, \rho \rangle \vdash p_1$ *and* $\langle C, \rho \rangle \vdash p_2 \Rightarrow \langle C, \rho \rangle \vdash p_1 \wedge p_2$

6. *Let* $\langle C, \rho \rangle \longrightarrow \langle C_i, \rho_i \rangle, 1 \leq i \leq n$, *be all the valid one step transitions from* $\langle C, \rho \rangle$. *Let* $\langle p, t \rangle \in OBS \times Term$ *be such that all variables occurring in p occur in* $\langle C, \rho \rangle$. *Furthermore, let* $\langle C_i, \rho_i \rangle \vdash \langle t, p \rangle$. *Then,* $\langle C, \rho \rangle \vdash p$

There are a number of points worth mentioning.

1. *Handling non-successful computations*
   Unsuccessful computations have been made observable. With deadlocked computations, the bindings in the environment can be observed. Infinite computations fall naturally in our framework. Even if a computation fails to terminate, bindings in the environments can be observed at intermediate stages of computation. A failed computation is interpreted as imposing an inconsistent constraint on the environment. However, note that only terminated and non-terminated computations can be distinguished. There are no mechanisms to distinguish deadlocked and infinite computations.

2. *Total correctness approach*
   The observables have the flavor of total correctness reasoning. This is because we demand that every valid computation sequence satisfies the predicate given by the observables.

The notion of tests depends on the contexts that can be used by the interpreter to differentiate the given programs. Let $C[]$ be a program definition, with a hole. The hole corresponds to a predicate whose definition is unknown. We say that $C[]$ is a valid context for a given program p if the predicate symbols used in the definitions of $C[]$ and p are disjoint. Thus, our notion of context is related to the notion of composable logic modules [3].

4 Domain-theoretic facts

In this section, the domain-theoretic tools needed for the semantics are sketched in an abstract setting.

4.1 Domain of values

The presentation of this subsection follows previous work [9]. To define the domain of terms we use a standard construction for defining a domain of (possibly infinite) terms in logic programming, see, for example, Lloyd [12]. First we need some notation. Let ω be the set of natural numbers. We use ω^* for the set of finite sequences of integers. A sequence is written $[i_1, \ldots, i_n]$. If s and t are sequences then $[s,t]$ denotes their concatenation, if s is a sequence and n is a natural number then $[s,n]$ is the sequence s with n added to the end. The size of a set X is witten $|X|$ and the size of a sequence s is written $|s|$.

Definition 6 *A* tree T *is a subset of* ω^* *satisfying*

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

2. $|\{i | [s,i] \in T\}|$ *is finite for all* $s \in T$.

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

The domain V is defined in two stages. First we define a domain W and then we add a top element, written T. The domain W is defined as follows. Let *Atom* be a given set of function symbols. Let $A = \mathit{Atom} \cup \{\Omega\} \cup \{f_i^n\}$ where Ω stands for the undefined element.

Definition 7 *An element of W is a function $F : t \to A$ where t is a non-empty tree. The function f satisfies*
$\forall s \in t.br(s) = n \Rightarrow F(s) = f$, where f is some arbitrary n-ary functional symbol, Ω treated as a 0-ary functional symbol.

The ordering between elements of W is defined as follows: $F \sqsubseteq G$ iff

- *$dom(F) \subseteq dom(G)$*
- *$\forall s \in dom(F).F(s) \neq \Omega \Rightarrow G(s) = F(s)$*

The ordering between elements of W allows one to replace occurrences of Ω with other elements to obtain a larger element. This domain describes infinitely deeply nested terms but all terms must have finite "width". Note that if two terms have different main function symbol, they are incomparable. Thus the domain decomposes into subdomains corresponding to different main function symbols. We denote the subdomain corresponding to the primary fucntion symbol f by W_f. If f is an n-ary function symbol, note that

$$x \in W_f \Rightarrow x = \perp \vee (\exists a_1 \ldots a_n \in W)\,[x = f(a_1 \ldots a_n)]$$

V is got from W by adding a top element denoted T. T is the model for inconsistent constraints. It is straightforward to check that the domain V is algebraic and consistently complete. Actually, V is an algebraic lattice.

In the sequel, we will be interested in algebraic lattices in which T is a finite element. The operational meaning of this assumption is the finite detectability of inconsistent constraints. Note that the domain V satisifes this property. We define below the T-strict product domain of algebraic lattices with finite T, D_1 and D_2. This is denoted $D_1 \times^T D_2$. The definition resembles the usual product structure except that, the top elements are "coalesced", so that the pairing operator is strict with respect to T. We denote the infinite T-strict product of D_i by $\Pi_i^T D_i$. An example of such an infinite product is the space of semantic environments $ENV = \Pi_x^T V_x$, where x ranges over variable names. The motivation for the definition of a T-strict product, is the need to propagate the error results of computations. For example an environment is inconsistent if any of the variables are bound to inconsistent values.

Definition 8 *Let D_1, D_2 be algebraic lattices with finite- T. $D_1 \times^T D_2$ is the partial order defined as follows:*

- *The elements of $D_1 \times^T D_2$ are :*
  - *Tuples $\langle d_1, d_2 \rangle$, where $d_i \in D_i \wedge \;\wedge\; d_i \neq T$*
  - *A element T*
- *The ordering relation is defined as follows:*
  - *T is the top element of the partial order.*
  - *The ordering between two tuples is the usual pointwise ordering.*

It is easy to check that $D_1 \times^T D_2$ is an algebraic lattice with T as a finite element. Note that $D_1 \times^T D_2$ is the cartesian product (in the categorical sense) in the category of algebraic lattices with finite T and T-strict continuous maps. So, given T-strict continuous maps $f_1 \in D_1 \to D'_1, f_2 \in D_2 \to D'_2$, we can define the map $f_1 \times^T f_2 \in D_1 \times^T D_2 \to D'_1 \times^T D'_2$ in the natural manner. The construction $\times^T$ is easily generalised to handle infinite products.

Definition 9 *$\mathbf{2}$ is the domain with two elements $\perp, T$, ordered as $\perp \sqsubseteq T$.*

5 Function spaces

5.1 Closure Operators

We first define closure operators on an algebraic lattice with finite top D. The definition is a variant of the defintion of closure operators [21]. For motivation, note that imposing a constraint on the environment increases the information in the environment. Furthermore, imposing a constraint twice is equivalent to imposing the constraint once.

Definition 10 *Let D be an algebraic lattice with finite T. $[(D) \xrightarrow{c} (D) \times \mathbf{2})]$ is the set defined as follows. The elements f of $[(D) \xrightarrow{c} (D) \times \mathbf{2})]$ are continuous functions from D to $D \times \mathbf{2}$, such that*

$$f(x) = \langle y, t \rangle \Rightarrow [x \sqsubseteq y \wedge f(y) = \langle y, t \rangle]$$

Note that the elements f of $[(D) \xrightarrow{c} (D) \times \mathbf{2})]$ are "idempotent" and "extensive". The second component of the result is intended to keep track of the termination of the computation.

A couple of examples will help to make the idea of closure operators clearer. Both the examples given below are used in the semantics later.

- Let f be a syntactic function symbol of arity n. We define a closure operator $\mathcal{E}[\![f]\!]$ on $\Pi_i^T V_i$, where $1 \leq i \leq (n+1)$. The intuition is that the first n arguments are approximations to the n argument places of the function. The last argument is an approximation to the final result. $\mathcal{E}[\![f]\!]$ is defined by cases as follows:

$$\begin{aligned}
\mathcal{E}[\![f]\!]\vec{a} &= \langle a_1 \ldots a_n, f(a_1 \ldots a_n) \ if \ a_{(n+1)} = \perp \\
\mathcal{E}[\![f]\!]\vec{a} &= T \ \ if \ a_{(n+1)} \notin W_f \\
\mathcal{E}[\![f]\!]\vec{a} &= \langle a_1 \ lubb_1 \ldots a_n \bigsqcup b_n f(a_1 \ldots a_n) \bigsqcup a_{(n+1)} \ \ otherwise
\end{aligned}$$

- Let D_1, D_2 be algebraic lattices with finite T. Let f be an element of $[(D_1) \xrightarrow{c} (D_1) \times \mathbf{2})]$. $f_1 \times^T Id_{D_2} \in [(D_1 \times^T D_2) \xrightarrow{c} (D_1 \times^T D_2) \times \mathbf{2})]$ is defined as :

$$(f_1 \times^T Id_{D_2})\langle x, y \rangle = \langle \langle \Pi_1(f_1(x)), y \rangle, \Pi_2(f_2(x)) \rangle$$

  Thus the computation of $f_1 \times^T Id_{D_2}$ on $\langle x, y \rangle$ terminates if and only if the computation of $f_1(x)$ terminates, as Id_{D_2} terminates always.

Let f be an closure operator on D. f can be equivalently characterised in terms of the "fixpoint" set. The "fixpoints" of f are elements x of D such that $f(x) = \langle x, t \rangle$. Define

$$Fix(f) = \{\langle x, t \rangle | f(x) = \langle x, t \rangle\}$$

Let $S \subseteq D \times \mathbf{2}$. Denote by S_x, the subset of elements of S of form $\langle y, t \rangle$ such that $x \sqsubseteq y$. Note that S_x might be empty.

Definition 11 *Let D be an algebraic lattice with finite T. A subset S of $D \times \mathbf{2}$ is said to be a valid set of fixpoints if it satsifies:*

1. *S is closed under least upper bounds of directed sets.*
2. $\langle x, t_1\rangle \in S \wedge \langle y, t_2\rangle \in S \wedge x \sqsubseteq y \Rightarrow t_1 \sqsubseteq t_2$
3. $(\forall x \in D)\,[\sqcap S_x \in S]$.

It is easy to see to see that the fixpoint set of a closure operator is a valid set of fixpoints. Furthermore, given a valid set of fixpoints, we can recover the coresponding closure operator. Let S be a valid set of fixpoints. Then, the corresponding partial closure operator f_S is defined as follows. Let $x \in D$. Then,

$$f(x) = \sqcap S_x$$

Now, we define a parallel composition operation $||$.

Definition 12 *Given $f, g \in [(D) \xrightarrow{c} (D) \times \mathbf{2})]$, $h = f||g$ is the closure operator defined as follows. Let $x \in D$. Let $y = \bigsqcup_i(((\Pi_1 \circ g) \circ (\Pi_1 \circ f)))^i(x))$, so that, $f(y) = \langle y, t_1\rangle, g(y) = \langle y, t_2\rangle$. Define,*

$$h(x) = \langle y, t_1 \sqcap t_2\rangle$$

Lemma 3 *$h = f||g$ defined as above is a member of $[(D) \xrightarrow{c} (D) \times \mathbf{2})]$.*

The proof is easy and is omitted.

Note that $h(x)$ is the least solution of a system of equations as follows:

$$h(x) \quad = \quad \text{lcs} \left\{ \begin{array}{l} x \sqsubseteq y \\ \langle y, t_1\rangle = f(y) \quad \textit{otherwise} \;\; \textit{in} \; \langle y, t_1 \sqcap t_2\rangle \\ \langle y, t_2\rangle = g(y) \end{array} \right.$$

This yields an elegant characterisation of the fixpoint set of $f||g$ in terms of the fixpoint sets of f and g. The proof follows easily from the the definition of $||$.

Lemma 4 $\langle x, t\rangle \in Fix(f||g) \Leftrightarrow (\exists \langle x, t_1\rangle \in Fix(f))\,(\exists \langle x, t_2\rangle \in Fix(g))\,[t = t_1 \sqcap t_2]$

It is immediate from the above lemma that the parallel composition operation has the expected desired properties.

Lemma 5 *$||$ is commutative, associative.*

Thus we can write the parallel composition of n elements $f_1 \ldots f_n$ without ambiguity as $||\{f_1 \ldots f_n\}$.

For the denotational semantics, we will need a variant of the parallel composition operator. Let $D_0, D_1 \ldots D_n$ be algebraic lattices with finite T. Let $f_i \in [(D_0 \times^T D_i) \xrightarrow{c} (D_0 \times^T D_i) \times \mathbf{2})]$. We define the "shared" parallel composition of $f_1 \ldots f_n$ as a closure oeprator on $D_0 \times^T D_1 \times^T \ldots D_n$. The intuition is that D_0 is "shared" among the f_i's. This operation arises in the semantics because the environment is shared among the processes. Roughly speaking, the meanings of processes will be closure operators on $ENV \times^T V$. To compute the parallel compostion of two such processes r_1, r_2, we need to ensure that both r_1 and r_2 "see" the same environment. We write this as $||_{D_0}\langle f_1 \dots f_n\rangle$. Let Id_i denote the identity function on D_i. Define

$$f'_i = (\Pi_0 \circ f_i \circ \langle \Pi_0, \Pi_i \rangle) \times^T Id_1 \dots (\Pi_i \circ f_i \circ \langle \Pi_0, \Pi_i \rangle) \times^T Id_n$$

Then, $||_{D_0}\langle f_1 \dots f_n\rangle = ||\{f'_1 \dots f'_n\}$.

We make $[(D) \xrightarrow{c} (D) \times 2)]$ into a partial order by defining an order relation on D.

$$f \sqsubseteq g \Leftrightarrow (\forall x \in D)\, [f(x) \sqsubseteq g(x)]$$

.

The order relation can be expressed in terms of the fixpoint sets.

Lemma 6 *Let $f, g \in [(D) \xrightarrow{c} (D) \times 2)]$. Then*

$$f \sqsubseteq g \Leftrightarrow (\exists \langle x, t_2 \rangle \in Fix(g)) \Rightarrow (\exists \langle x, t_1 \rangle \in Fix(f))\, [t_1 \sqsubseteq t_2]$$

Proof: Let $\langle x, t \rangle \in Fix(g)$. Then, $f(x) \sqsubseteq g(x)$. Note that $g(x) = \langle x, t_2 \rangle$. Hence, $f(x) = \langle x, t_1 \rangle$ and the forward implication follows.

For the reverse implication, note that the condition implies

$$(\forall x \in D)\, [Fix(g)_x \neq \phi \Rightarrow (Fix(f)_x \neq \phi \wedge \sqcap(Fix(f)_x) \sqsubseteq \sqcap(Fix(g)_x))] \qquad \blacksquare$$

For the denotational semantics we need a notion of guarded closure operators. Let $Bool == \{\perp, tt, ff, T\}$ be the lattice of truth values. Let s be a continuous function from ENV to $Bool$. We can define a continuous operator $s|$ on $[(D_0) \xrightarrow{c} (D_0) \times 2)]$. We write this infix. Let $f \in [(D_0) \xrightarrow{c} (D_0) \times 2)]$. Then, $s|f$ is defined as

$$(s|f)v = \begin{cases} f(v) \text{ if } s(v) = tt \\ \langle T, T \rangle \text{ if } ff \sqsubseteq s(v) \\ \langle v, \perp \rangle \text{ otherwise} \end{cases}$$

The monotonicity and the continuity of the above are easy to check.

Finally, the following lemma enables us to set up the usual cpo framework for the semantics of recursion.

Lemma 7 *Let D be an algebraic lattice with finite T. Then, the space $[(D) \xrightarrow{c} (D) \times 2)]$ is an algebraic lattice.*

Proof: (Sketch)
Let f, g be elements of $[(D) \xrightarrow{c} (D) \times 2)]$. The least upper bound of h of f and g is defined as

$$h(x) = \text{lcs} \begin{cases} x \sqsubseteq y \\ \langle y, t_1 \rangle = f(y) \langle y, t_2 \rangle = g(y) \end{cases} in\ \langle y, t_1 \sqcup t_2 \rangle$$

Note the close similarity between the above definition and the definition of the parallel composition operation. Least uuper bounds of chanis are got by the usual pointwise limit, and the finite elements are lubs of finite sets of functions of the form $f_{a,b}$, where $a \sqsubseteq b$ are finite elements and defined by

$$f_{a,b}(x) = \begin{cases} x \text{ if } a \not\sqsubseteq x \\ x \sqcup b \text{ otherwise} \end{cases} \qquad \blacksquare$$

5.2 Powerdomain constructions

The modelling of indeterminacy requires machinery to handle sets of closure operators. The development of these tools is is the primary aim of this section. The powerdomain construction that is used here is a variant [8] of the standard powerdomain constructions [23]. The relationship of the powerdomian construction used here to the usual powerdomian constructions is described in Appendix D.

Let $B(D)$ be the basis of D. The basis of the Smyth powerdomain of D, denoted by $P_S(D)$ is the finite powerset of the basis elements, $P_{fin}(B(D))$, ordered as

$$\{d_1 \ldots d_n\} \sqsubseteq \{e_1 \ldots e_m\} \Rightarrow (\forall 1 \leq j \leq m)\ (\exists 1 \leq i \leq n)\ [d_i \sqsubseteq e_j]$$

The Smyth powerdomain [23] of D, denoted $\overline{P_S(D)}$, is the ideal completion of $P_S([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])$, i.e we have

- The elements of $\overline{P_S(D)}$ are downward closed and directed subsets S of $P_S(D)$
- The ordering relation is $\subseteq$.

$\overline{P_S(D)}$ can be made into a continuous algebra [23] with a union operation $\uplus$, defined as follows:

$$S_1 \uplus S_2 = \{s_1 \bigcup s_2 | s_1 \in S_1, s_2 \in S_2\}$$

Note that the operation $\uplus$ is idempotent, commutative and associative.

The new powerdomain construction is based on an "extenstional" ordering among sets of functions. Define a set-theoretic function *App*, from $P_{fin}(B([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])) \times D$ to $\overline{P_S(D)}$ as follows.

$$App(\{f_1 \ldots f_n\}, x) = \uplus_i f_i(x)$$

The ordering relation is defined in terms of the partial function *App*.

Definition 13 *$P([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])$ is the preorder whose carrier is $P_{fin}(B([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})]))$. The ordering relation is defined as follows.Let $F = \{f_1 \ldots f_n\}, G = \{g_1 \ldots g_m\}$. Then $F \sqsubseteq G$ if,*

$$F \sqsubseteq G \Rightarrow [(\forall x)\ App(F, x) \sqsubseteq App(G, x)]$$

$\overline{P([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$ is the ideal completion of $P([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])$. $\overline{P([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$ can be made into a continuous algebra [23] with a union operation $\uplus$, defined as follows:

$$S_1 \uplus S_2 = \{s_1 \bigcup s_2 | s_1 \in S_1, s_2 \in S_2\}$$

. $\uplus$ is idempotent, commutative and idempotent.

There is a singleton embedding function $\{\!|\ |\!\}$ from $[(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})]$ to $\overline{P([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$, defined on the basis of $[(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})]$ by

$$\{\!|\ f\ |\!\} = \{f\}$$

. $\{\!|\ |\!\}$ as defined above is monotone, and can be extended uniquely to a continuous function.

The parallel composition operation on $\overline{P([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$ can be defined, in a natural fashion. The following definition is on the basis elements $P([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])$.

$$\{f_1 \ldots f_n\} || \{g_1 \ldots g_m\} = \uplus \{f_i || g_j | i, j\}$$

Lemma 8 $||$ *as defined above is monotone in both arguments.*

Proof: Let $F = \{f_1 \dots f_n\}$, $G = \{g_1 \dots g_m\}$ and $G \sqsubseteq F$. Let $H = \{h_1 \dots h_k\}$. Let $\langle y, term\rangle \in App(F||H, x)$. Then, we have $f_i \in F, h_k \in H$ such that $f_i||h_k(x) = \langle y, term\rangle$. Since f_i and h_k are finite, there is an i such that

$$\begin{array}{rcl} y & = & (((\Pi_1 \circ f_i) \circ (\Pi_1 \circ h_k)))^i(x)) \\ f_i(y) & = & \langle y, t_1\rangle \\ h_k(y) & = & \langle y, t_2\rangle \\ term & = & t_1 \sqcap t_2 \end{array}$$

Since $G \sqsubseteq F$ $(\exists g_j)$ $[g_j(y) \sqsubseteq f_i(y)]$. Thus $g_j(y) = \langle y, t_1'\rangle$, for some $t_1' \sqsubseteq t_1$. Thus

$$g_j||h_k(y) \sqsubseteq f_i||h_k(y)$$

Since $x \sqsubseteq y$,

$$g_j||h_k(x) \sqsubseteq g_j||h_k(y) \sqsubseteq f_i||h_k(y) = \sqsubseteq f_i||h_k(x)$$

Thus, $App(G||H, x) \sqsubseteq App(F||H, x)$. ∎

So, $||$ can be extended uniquely to a continuous function on the whole space. The $||$ operation is commutative and associative. As before, the parallel composition operation with sharing can also be defined. The definition of the operation of parallel composition with sharing, requires a notion of $F \times^T Id_{D_2}$, $F \in P([(D_2) \xrightarrow{c} (D_2) \times \mathbf{2})])$. We define a continuous function $\times^T Id_{D_2}$. The domain of $\times^T Id_{D_2}$ is $P([(D_1) \xrightarrow{c} (D_1) \times \mathbf{2})])$ and its range is $P([(D_1 \times^T D_2) \xrightarrow{c} (D_1 \times^T D_2) \times \mathbf{2})])$. This is written postfix for readability. As usual, we define $s \times^T Id_{D_2}$, for $s \in P([(D_1) \xrightarrow{c} (D_1) \times \mathbf{2})])$. Let $s = \{f_1 \dots f_n\}$.

$$s \times^T Id_{D_2} = (f_1 \times^T Id_{D_2}) \uplus \dots (f_n \times^T Id_{D_2})$$

It is easy to check that the above definition defines a monotone function from $P([(D_2) \xrightarrow{c} (D_2) \times \mathbf{2})])$ to $P([(D_1 \times^T D_2) \xrightarrow{c} (D_1 \times^T D_2) \times \mathbf{2})])$. So, it can be extended uniquely to a continuous function from $P([(D_1) \xrightarrow{c} (D_1) \times \mathbf{2})])$ to $P([(D_1 \times^T D_2) \xrightarrow{c} (D_1 \times^T D_2) \times \mathbf{2})])$.

As in the determinate case, we can define a continuous opertor on $s|$ on $P([(D) \xrightarrow{c} (D) \times \mathbf{2})])$, where s is a continuous function from ENV to $Bool$. As usual, we define s only on the finite elements of $P([(D) \xrightarrow{c} (D) \times \mathbf{2})])$.

$$s|\{f_1 \dots f_n\} = \uplus_i \{s|f_i\}$$

Monotonicity is easily checked. Furthermore, the function defined above preserves $\uplus$, i.e is linear. So, it can be extended uniquely to a continuous, linear operator on $P([(D) \xrightarrow{c} (D) \times \mathbf{2})])$.

6 Denotational Semantics

The types of the denotations of various syntactic entities is as follows:

- Terms : $[(ENV \times^T V) \xrightarrow{c} (ENV \times^T V) \times \mathbf{2})]$.
- n-ary predicates p : element of $P([(\Pi^T_{1 \le i \le n} V_i) \xrightarrow{c} (\Pi^T_{1 \le i \le n} V_i) \times \mathbf{2})])$

- Atoms and sequences of atoms: $\overline{P([(ENV) \xrightarrow{c} (ENV) \times 2])}$.

The definition of the denotations of terms is by structural induction on terms. The definition will encompass meanings of sequences of terms, $\mathcal{E}[\![\langle t_1 \ldots t_n \rangle]\!]$. The denotation of a sequence of terms of length n is a closure operator on the space $ENV \times^T \Pi_n^T V$. The intuitive way to read the definition is to treat the ENV argument to the function as the environment of evaluation of the term, and the V argument to the function as the approximation to the final result of evaluating the term.

- Variables :

$$\mathcal{E}[\![x]\!] \ \langle env, a \rangle = \langle \langle env[x \mapsto b], b \rangle, T \rangle \ \ where\ b = env(x) \bigsqcup a$$

- Sequences of terms:
Let $\vec{t} = \langle t_1 \ldots t_n \rangle$. Note that by structural induction hypothesis, $f_i = \mathcal{E}[\![t_i]\!]$ are defined. Then,

$$\mathcal{E}[\![\langle t_1 \ldots t_n \rangle]\!] = ||_{ENV} \langle \mathcal{E}[\![t_i]\!] | i \rangle$$

- Terms:
Consider a term of form $g(t_1 \ldots t_n)$. By structural induction hypothesis, $\mathcal{E}[\![\vec{t}]\!]$ is known. Define,

$$\mathcal{E}[\![g(t_1 \ldots t_n)]\!] = \langle \Pi_{ENV}, \Pi_{(n+1)} \rangle \circ ((\mathcal{E}[\![\vec{t}]\!] \times^T Id_{V(n+1)}) || (Id_{ENV} \times^T \mathcal{E}[\![g]\!]))$$

Now we define the meaning of the equality predicate, as a closure operator on ENV. The denotation of $t_1 = t_2$ is a closure operator on ENV. The result of evaluating $t_1 = t_2$ in an environment is the environment got by adding this constraint. In keeping with the spirit of constraints, the resulting environment can be thought of as the smallest environment more refined than the input environment such that both t_1 and t_2 evaluate to the same value. Since least common solutions were captured by the parallel composition operation, the following definition should not be surprising.

$$\mathcal{E}[\![t_1 = t_2]\!](env) = [\Pi_{ENV} \circ (\mathcal{E}[\![t_1]\!] || \mathcal{E}[\![t_2]\!])] \ \langle env, \vec{\bot} \rangle)$$

Thus the resulting environment is got by applying the parallel composition of $\mathcal{E}[\![t_1]\!]$ and $\mathcal{E}[\![t_2]\!]$ to the initial environment, and projecting out the resulting environment.

The denotation of a sequence of clauses C is built up by induction on length. The following definition builds up the denotations of larger sequences from smaller sequences.

$$\mathcal{E}[\![C_1, C_2]\!] = \mathcal{E}[\![C_1]\!] || \mathcal{E}[\![C_2]\!]$$

Under AND-fairness, note that the constraints imposed by C_1, C_2 is the intersection of the constraints imposed by the C_i's. The denotational semantics models this view of the operational semantics, with intersection of constraints being modelled by the parallel composition operator $||$. Furthermore, the computation corresponding to the sequence of atoms above terminates exactly when both the subcomputations C_1 and C_2 terminate. Note that this was built into the domain theoretic definition of the parallel composition operator: the greatest lower bound oepration in defining the termination signal of the result of the parallel composition operation in definition 12 captures exactly this notion.

We now define the meaning of guarded clauses of the form $G|C$. This requires the definition of the denotation of guards. The denotation of guards is a continuous function mapping the space of environments ENV to the lattice of truth values $Bool$. Let G be a guard predicate. Let env be an element of ENV. Then, $\mathcal{E}[\![G]\!]\ env = tt$ is intended to mean that the guard evaluates to true in the environment env. Similarly, $\mathcal{E}[\![G]\!]\ env = ff$ is intended to mean that the guard evaluates to true in the environment env, and $\mathcal{E}[\![G]\!]\ env = \perp$ is intended to mean that the evaluation of the guard in the environment env leads to a suspended computation. The formal definition is given below.

$$\mathcal{E}[\![x \uparrow s = f]\!]env = \begin{cases} T, \text{ } if \text{ } env = env_T \\ tt, \text{ } if \text{ } env(x) \uparrow s = f \\ ff, \text{ } if \text{ } env(x) \uparrow s = g \neq f \\ \perp \text{ } otherwise \end{cases}$$

The conjunction of a list of primitive guards is defined using the "parallel AND" function, defined as follows. Let f_1, f_2 be continuous functions from ENV to $Bool$.

$$Pand(f_1, f_2)env = \begin{cases} T, \text{ } if \text{ } env = env_T \\ ff, \text{ } if \text{ } f_1 \text{ } env = ff \vee f_2 \text{ } env = ff \\ tt, \text{ } if \text{ } f_1 \text{ } env = tt \wedge f_2 \text{ } env = tt \\ \perp \text{ } otherwise \end{cases}$$

Define

$$\mathcal{E}[\![G|C]\!] = \mathcal{E}[\![G]\!]|\mathcal{E}[\![C]\!]$$

This is motivated by considering the three cases of the try function for the guard predicate:

- (Success)
  This happens operationally when the environment has sufficient constraints to make the guard predicate succeed. Semantically, this is modelled by checking that $\mathcal{E}[\![G]\!]$ in the environment returns true. Recall $\mathcal{E}[\![G]\!]|\mathcal{E}[\![C]\!]$ returned the result of $\mathcal{E}[\![C]\!]$ if the input environment was more refined than $\mathcal{E}[\![G]\!]$.

- Failure
  This happens operationally when the environment has sufficient constraints to make the guard predicate fail. Semantically, this is modelled by checking that $\mathcal{E}[\![G]\!]$ in the environment evaluates to false. Recall that $\mathcal{E}[\![G]\!]|\mathcal{E}[\![C]\!]$ returned T if the input environment was such that $\mathcal{E}[\![G]\!]$ evaluated to ff.

- Suspend
  This happens operationally when neither of the above happens. In this case, this branch of computation suspends. Semantically, this is modelled by returning the input environment as the resulting environment, and keeping track of the fact that computation has not terminated.

The denotation of p is an element of $[(\Pi^{T}_{1 \leq i \leq n} V_i) \xrightarrow{c} (\Pi^{T}_{1 \leq i \leq n} V_i) \times \mathbf{2})]$. For motivational purposes, consider the simple case when there is only one non-recursive definition for the 1-ary predicate p. Let the definition be $p(x) \leftarrow G|C$. Define

$$\mathcal{E}[\![p]\!] = \lambda a.\Pi_x \circ \mathcal{E}[\![G|C]\!] \circ env_{\perp}[x \mapsto a]$$

The handling of renaming by the above definition needs some explanation. Intuitively, the above definition can be viewed as setting up a new local environment for the execution of p, and throwing it away when execution of p is done. This is brought out by the following observations:

- The environment passed to $\mathcal{E}[\![G|C]\!]$ is uninitialised except for the variable x.
- The resulting environment is thrown away at the end, and only the value of the environment at x is returned as the resulting semantic value.

Now, consider the general case. The meanings of n-ary predicates are elements of the powerdomain of the closure operators on $\Pi_n^\top V$. The elements of $\mathcal{E}[\![p]\!]$ correspond to the different choices of execution. Furthermore, each element(choice) has the same intutive reading that we gave above. We pass approximations to the arguments to $\mathcal{E}[\![p]\!]$ and $\mathcal{E}[\![p]\!]$ returns a set of refined results corresponding to the different possible execution paths.

Let the defining clauses for p be given by

$$p(\vec{x}) \leftarrow G_1|C_1$$
$$\vdots$$
$$p(\vec{x}) \leftarrow G_m|C_m$$

Because of recursion, some of the atoms in some C_i might have predicate name p.

We present the motivation for the definitions first.

- Handling many clauses with same head:
  Given a set of clauses for p, we have different possible choices for expanding the occurrence of a p in a goal. Thus p can be viewed as imposing one of a set of constraints. Note that the $\uplus$ operation of the powerdomain constructions models this presence of choice.
- Recursion :
  Handling of recursion is standard. The correspondence between least fixed points and recursive definitions is well known and has been widely studied, in various settings.

The denotation of p, $\mathcal{E}[\![p]\!]$, is defined by as the least fixed point of a functional τ. τ is of type $P_S([(V^n) \xrightarrow{c} (V^n) \times 2])]) \rightarrow P_S([(V^n) \xrightarrow{c} (V^n) \times 2])])$. τ is defined as follows:

$$\tau(f) = \uplus_i\{\mathcal{E}[\![\tau_i(f)]\!] | 1 \leq i \leq m\}$$

where, $\mathcal{E}[\![\tau_i(f)]\!]$ is defined as follows:

$$\mathcal{E}[\![\tau_i(f)]\!] = \lambda\vec{a}.\Pi_{\vec{x}} \circ \mathcal{E}[\![G_i|C_i]\!] \circ [env_\perp[\vec{x} \mapsto \vec{a}]]$$

Note that $\tau_i(f)$ depends on f if there is an occurrence of p in C_i.

A notion of application is defined next, i.e the denotation of $p(\vec{u})$ assuming that $\mathcal{E}[\![p]\!]$ is known. This semantic function is intended to model the result got by executing the goal query $p(\vec{u})$ in an environment. This is the base case for the denotation of clauses. The definition is motivated by studying the operational rule for the reduce transition. Informally, the reduce transition has the following components:

- A renamed apart definition of p.

- A unifier that binds some of the renamed apart variables to compnents of the term $\vec{u}$.

The first item above was modelled by $\mathcal{E}[\![p]\!]$. The effects on the environment is only through the unifier in the second step. The connection between the local environment of p and the global environment is that the result of evaluating $\vec{u}$ in the global environment is the same as the result of evaluating the relevant vector of local variables in the local environment. Define

$$\mathcal{E}[\![p(\vec{u})]\!] = \Pi_{ENV} \circ (Id_{ENV} \times^T \mathcal{E}[\![p]\!]) || (\{\!| \mathcal{E}[\![\vec{u}]\!] |\!\}) \circ \langle Id_{ENV}, \vec{\bot} \rangle$$

Note that the global environment is affected only by $\mathcal{E}[\![\vec{u}]\!]$. Also, the result of evaluating the relevant vector of local variables in the local environment of p was the result returned by p. Thus, the parallel composition operation (read as common solution) ensures that the "value" resulting from evaluating $\mathcal{E}[\![\vec{u}]\!]$ in the global environemnt and the value returned by execution of p are equal.

7 Relating the two semantics

In this section, we outline the proof that the operataional and denotational views of programs coincide. Here, we restrict ourselves to an informal description of the key ideas underlying the proof. Formal definitions and detailed proofs can be found in the appendices. The proof itself can be viewed as the extension of the full-abstraction proof of previous work [9] to an indeterminate setting.

Reduction preserves meaning

The first step in a full-abstraction proof is a soundness result of the denotational semantics. The aim of this is to show that the denotation of a program remains invariant during reduction. In the setting of determinate languages, this soundness result is proved by showing that one step of the reduction relation does not alter the denotation of the program. In an indeterminate setting a reduction step could involve making a choice among competing and mutually exclusive reductions. Thus, it is unrealistic to expect such a result in the indeterminate setting.

We first need to associate denotations with operational configurations. The syntactic environment can be translated into a set of equations in the following way. Let A be an alias set. Then, $EQ(A)$ is the set of all pairs of terms in the alias set. $EQ(\rho)$, the set of equations generated from ρ, is the reflexive, transitive and symmetric closure of the union of the equations generated from each alias set in ρ.. The semanticfunction $\mathcal{M}[\![]\!]$ assigns to configurations an element of $P([(ENV) \stackrel{c}{\rightarrow} (ENV) \times \mathbf{2})])$.

$$\mathcal{M}[\![\langle C, \rho \rangle]\!] = \mathcal{E}[\![C]\!] || \mathcal{E}[\![EQ(\rho)]\!]$$

Thus, it is intended that $\mathcal{M}[\![]\!]$ represents the effect of the complete computation on a configuration. $\mathcal{M}[\![\langle C, \rho \rangle]\!]$ evaluated in an initial environment yields a set of possible results, say S. Let $conf_1 \ldots conf_n$ be all the possible configurations reachable in one-step from $\langle C, \rho \rangle$. Consider the sets S_i, where S_i is the set of possible results got by evaluating $\mathcal{M}[\![conf_i]\!]$ in the same initial environment. Then, we show that $S = \uplus_i S_i$. As a corollary, we show if a process passes a finite test, the set of possibel results produced by the denotational semantics attests this fact.

Computational adequacy

The hardest part of the proof is the converse to the result stated at the end of the last subsection, namely that a process passes a test *only if* the results predicted by the denotational semantics indicate so. This is proved by showing that the operational semantics attains the values predicted by the denotational semantics. More precisely, it is shown that the operational semantics attains every *finite approximant* to the result predicted by the denotational semantics. Analagous properties have been termed computational adequacy [13].

For this, we define a relationship $\preceq$ between sequences of atoms C and elements G of the powerdomain of closure operators on ENV. The main theorem proves that for all sequences of atoms C, $\mathcal{E}[\![C]\!] \preceq C$. Informally, $\mathcal{E}[\![C]\!] \preceq C$ means the following. Recall that $\mathcal{E}[\![C]\!]$ can be thought of intuitively as a set of closure operators. $\mathcal{E}[\![C]\!] \preceq C$ means that every valid computation sequence c of C corresponds to an element f_c of $\mathcal{E}[\![C]\!]$. This correspondence takes the following form. Assume that we are given a finite piece of the result predicted by f_c. Then, c after a finite sequence of reductions produces a more refined value.

The proof proceeds by structural induction. The difficult part of the proof is the construction of a reduction sequences from semantic information. The subtle case is the handling of parallel imposition of constraints. The special properties of the parallel composition of closure operators enable us to carry out this construction. We make use of the fact that the semantic definition of the least common fixed point of a pair of closure operators suggests an interleaving strategy. Suppose that g_1 and g_2 are two closure operators that correspond to the imposition of two constraints given by sequences of atoms C_1 and C_2. Suppose that we know how to construct reduction sequences corresponding to C_1 and C_2 individually. Then, from the definition of the parallel composition g_1 and g_2, we can construct an interleaved reduction sequence of C_1 and C_2 corresponding to the computing the iterates of $(g_1 \circ g_2)$.

Full-abstraction

Combining the above two results, we deduce that a process passes a finite test, if and only if the set of possible results produced by the denotational semantics witnesses this fact. Since the denotational semantics is compositional, if two processes have the same denotation, the tests passed by one process are identical to the test passed by the second process. Thus, the denotational semantics is correct for resoning about operational equality. This is called adequacy [2].

In fact the converse is also true. If two processes do not have the same denotation, there is context that distinguishes the two processes, i.e the tests passed by the configuration got by placing one process in the context differs from the tests passed by configuration got by placing the other process in the same context. This is called full abstraction [16] [13].

8 Semantics of full language

This section extends the semantics to the full language. The syntactic differences arise from the more powerful tests allowed int he gaurd predicate. The primitive guards are enhanced to allow tests of the from $x = y$, where x, y are variable names. In general, the primitive guards can be of form $x = t$, where t is an arbitrary term.

In the semantics, this difference is reflected in the notion of observations. Primitive observations are now of the form $x = t, x \neq t$. A syntactic environment ρ models a primtive observation $x = t$ if $\rho \bigcup \{x, t\} = \rho$, i.e the reduced form of $\rho \bigcup \{x, t\}$ is ρ.

The denotational semantics has the same structure as before. The domains ENV and V are defined differently. However, as before, both these domains will be complete algebriac lattices with finite T. Thus, the original definitions of combinators like parallel composition are valid in this context. The only semantic functions that change are the definitions of the denotations of variables and guard predicates. There is also a slight difference in the definition of the denotation of predicate symbols. Thus the proof of the match of the operational and denotational semantics for the restricted language go through essentially unchanged.

The semantic domain of environments is constrcuted from the syntactic environments. Intutively, the ordering relation captures the notion of "more defined". So, $\rho_1 \sqsubseteq \rho_2$ is intended to mean that the constraints imposed by ρ_1 are a subset of the constraints imposed by ρ_2. This can be captured operationally by saying that the reduced from of $\rho_1 \bigcup \rho_2$ is ρ_2. Since the denotational semantics requires the presence of limits, the semantic domain ENV includes the limits of sequences of finite syntactic environments. Indeed, ENV can be viewed as the ideal completion of the space of finite syntactic environments oredered by the "refinement ordering". A more explicit description of the domain ENV is given below.

Definition 14 *ENV is the preorder defined as follows.*

- *The elements of ENV are (possibly) infinite syntactic enviroments, in reduced form.*
- $[\rho_2 \textit{ inconsistent} \vee \rho_2 \cup \rho_1 = \rho_1] \Rightarrow \rho_1 \sqsubseteq \rho_2$

ENV shares the nice properties of the space of environments used for the simpler language.

Lemma 9 *ENV is a complete algebraic lattice with finite T.*

Proof: (Sketch)
The finite elements are finite syntactic environments. The least upper bound of a set of syntactic environments $\{\rho_i | i \in I\}$, where I is an index set, is given by the reduced form of $\bigcup_i \rho_i$. The T element is the inconsistent environment. ∎

To define the space of values, we need a notion of the value of a term in a syntactic environment. Recall that this notion was used in defining the transition relation for the simpler language in section 2.

The space of values V is defined as follows.

Definition 15 *V is the preorder defined as follows.*

- *The elements of V are of the form $\langle t, \rho \rangle$, where t is a term and $\rho \in ENV$.*
- *$\langle t_1, \rho_1 \rangle \sqsubseteq \langle t_2, \rho_2 \rangle$, if ρ_2 is inconsistent or both of the following conditions hold:*
  - $\rho_1 \sqsubseteq \rho_2$
  - $(\forall s)\ [(t_1, \rho_1) \uparrow = f_n^i \Rightarrow (t_2, \rho_2) \uparrow = f_n^i]$

Lemma 10 *V is a complete algebraic lattice.*

Proof: (Sketch)
The finite elements are of the form $\langle t, \rho \rangle$, where ρ is a finite element of ENV. The T elements are of the form $\langle t, \rho \rangle$, where ρ is inconsistent. The least upper bound of a set $\{\langle t_i, \rho_i \rangle | I\}$ is given by $\langle t_1, \rho \rangle$, where ρ is the reduced form of $[\bigcup_i \rho_i] \bigcup \{t_i | I\}$. ∎

Thus, the parallel composition combinator can be defined for all the cases required in the semantics. Below, the definitions that differ from the semantics of the weaker language, are sketched. The other definitions take exactly the same form as before, and are omitted.

- Variables :
  Let $v = \langle t, \rho \rangle$. Then

  $$\mathcal{E}[\![x]\!]\ \langle env, a \rangle = \langle \langle env', b \rangle, T \rangle,\ where\ = env(x) \bigsqcup a$$

  where $env' = env \bigcup \rho \bigcup \{x = t\}$ and $b = \langle t, env' \rangle$.

- Guard predicates:
  The guard predicate is conjunction of primitive guards.

  - $x = t$, $x \uparrow s = f, x \uparrow s \neq f$, where f is a function symbol and x is a variable name and t is a term.
  - Various numeric predicates, for example $=:=, \leq, \neq:\neq$,

  The denotation of guards is continuous function from ENV to $Bool$ as before. The denotation of primitive guards is defined below.

  $$\mathcal{E}[\![x = t]\!]env = \begin{cases} T,\ if\ env = env_T \\ tt,\ if\ env \bigcup \{x, t\} = env \\ ff,\ if\ env \bigcup \{x, t\} = env_T \\ \perp\ otherwise \end{cases}$$

  The conjunction of a list of primitive guards is defined using the "parallel AND" function, defined as before.

- Predicate symbols :
  Recall that the original definition went as follows. The denotation of p, $\mathcal{E}[\![p]\!]$, is defined by as the least fixed point of a functional τ. τ is a continuous operator on the space $P([(V^n) \xrightarrow{c} (V^n) \times 2\!\!\!/)])$. τ is defined as follows:

  $$\tau(f) = \uplus_i \{\mathcal{E}[\![\tau_i(f)]\!] | 1 \leq i \leq m\}$$

  where, $\mathcal{E}[\![\tau_i(f)]\!]$ is defined as follows:

  $$\mathcal{E}[\![\tau_i(f)]\!] = \lambda \vec{a}.\Pi'_{(\vec{x},\vec{a})} \circ .\mathcal{E}[\![G_i | C_i]\!] \circ [env_\perp[\vec{x} \mapsto \vec{a}]]$$

  where $\Pi'_{(\vec{x},\vec{a})}$ is a continuous function from ENV to V defined as follows:

  $$\Pi'_{(\vec{x},\vec{a})} env = \langle (a_1, env(x_1) | r) \ldots (a_n, env(x_n) | r)$$

  where $env(x_1)|r$ is the alis set of x_1 in env with all occurrences of $\vec{x}$ removed. In the above definition, there is an implicit assumption that $\vec{x} \bigcap Var(\vec{a}) = \phi$. We can make this assumption because it suffices to define the value of $\mathcal{E}[\![\tau_i(f)]\!]$ on finite $\vec{a}$. If $\vec{a}$ is finite, the number of variables in $Var(\vec{a})$ is finite. So, the there are always unbound variables availabel for use as $\vec{x}$ in the above definition. Furthermore, note that the result returned is *independednt* of the choice of the variable names $\vec{x}$ as long as they satisfy the disjointness condition.

9 Conclusions

The semantics presented here presents a rather simple and straightforward view of programs. The setting works naturally for infinite and deadlocked computations, and the AND indeterminacy of the operational semantics is abstracted away totally. However, the treatment of error in computations as "benign" is not very satisfying: for example the following two programs are identified:

$$p(x) \leftarrow true|x = 1$$

and the predicate name q with two definitions

$$q(x) \leftarrow true|x = 1$$
$$q(x) \leftarrow true|fail$$

This issue is to be addressed in future work.

10 Acknowledgements

We wish to thank Prakash Panangaden and Keshav Pingali for invaluable discussions. Indeed, the strong influence of their work on functional langauges with logic variables [9] on this work is immediately obvious to anyone who has read their paper. Radha Jagadeesan would also like to thank V. A. Saraswat for helping him understand the framework of concurrent constraint logic programming, and listening to earlier presentations of this work, and pointing out the deficiencies. He would also like to thank Georges Lauri and Jayanti Prasad for useful comments and criticism during the development of this work.

References

[1] A. Colmerauer. Prolog and infinite trees. In Clark and Tarnlund, editors, *Logic Programming*, pages 231–251. Academic Press, New York, 1982.

[2] P-L Curien G. Berry G and J. J. Levy. The full-abstraction problem: the state of the art. In *Algebraic Methods in Semantics.* Cambridge University Press, 1985.

[3] M.J. Maher H. Gaifman and E. Shapiro. Reactive behaviour semantics for concurrent constraint languages. In E. Lusk and R. Overbeck, editors, *Proceedings of the North American Conference on Logic Programming.* MIT Press, 1989.

[4] M. Henessy and G. Plotkin. Full abstraction for a simple parallel programming language. In *Mathematical Foundations of Computer Science, Lecture notes in Computer Science 74*, 1979.

[5] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

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

[7] J. L. Lassez J. Jaffar and M. J. Maher. Prolog ii as an instance of a logic programming language scheme. In M. Wirsing, editor, *Proc. IFIP Conference.* North-Holland, 1987.

[8] R. Jagadeesan and P. Panangaden. A domain-theoretic model for a higher-order process calculus. In *Proceedings of The International Conference on Automata, Languages and Programming*, pages 181–194. Springer-Verlag, July 1990. LNCS 443, Cornell TR 89-1058.

[9] R. Jagadeesan, P. Panangaden, and K. Pingali. A fully-abstract semantics for a functional language with logic variables. In *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science*, pages 294–303, 1989.

[10] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 993–998. North-Holland, 1977.

[11] J. Kok. A compositional semantics for prolog. In *Proceedings of Symposium on Theoretical aspects of Computer Science*, pages 373–388. Springer-Verlag, 1988. LNCS 294.

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

[13] A. Meyer. Semantical paradigms. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, 1988.

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

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

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

[17] 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.

[18] V. Saraswat. The concurrent logic programming language cp: Definition and operational semantics. In *Proceedings of ACM Principles of Programming Languages*, pages 49–63, 1987.

[19] V. A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1988.

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

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

[22] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

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

[24] J. E Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, volume 1 of *The MIT Press Series in Computer Science*. MIT Press, 1977.

[25] K Ueda. Guarded horn clauses: A parallel programming langauge with the concept of a guard. *Programming of Future Generation computers*, 1988.

[26] M. Rinard V. Saraswat and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, January 1991.

[27] W. P. Weijland. Semantics for logic programs without occur check. *Theoretical Computer Science*, 71(1990):155–174, 1990.

A One-step Reduction Preserves Meaning

In this section we will show that the reduction relation preserves meaning, as given by the abstract semantics. In particular, this shows that if a sequence of rewrites resulst in a configuration that cannot be reduced any further then the constraints embodied in the configuration are predicted by the abstract semantics. It also means that the constraints of intermediate configurations are subsumed in the final result predicted by the environment. For this we need to translate the syntactic environment into a set of equations. We formalise this notion first.

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

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. Let A be an alias set. Then, $EQ(A)$ is the set of all pairs of terms in the alias set.

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 an element of $\mathcal{P}([(ENV) \xrightarrow{c} (ENV) \times 2])$.

$$\mathcal{M}[\![\langle C, \rho\rangle]\!] = C || \mathcal{E}[\![EQ(\rho)]\!]$$

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

$$Dom(env) \cap (Var - Dom(\rho) = \phi$$

$Dom(env)$ is the set of names that are bound to a non-bottom value. $Dom(\rho)$ refers to the set of all names occurring in some alias set. This restriction ensures that there will be no conflicts occurring when new names are allocated. The function $\mathcal{M}$, defines the meaning of sequences of atoms in the context of a syntactic environment ρ. Thus, it is intended that $\mathcal{M}[\![]\!]$ represents the effect of the complete computation on a configuration. The following theorem shows that, in a certain sense, as we rewrite a configuration the meaning as given by $\mathcal{M}$ will not alter. More precisely, we prove that the part of the environment that is initially relevant is preserved by the one-step reduction. The reason we need this restriction is that some of the rewrites may cause new variables to be generated; in that case one clearly cannot hope that the environments are identical. We use the notation $|_{bv(\rho)}$ to mean that the resulting environment is restricted to the variables that were bound in the environment ρ.

Lemma 11 *Let the defining clauses for p be given by*

$$p(\vec{x}) \leftarrow G_1 | C_1$$
$$\vdots$$
$$p(\vec{x}) \leftarrow G_m | C_m$$

Consider the configuration $\langle p(\vec{t}), \rho\rangle$. Then,

$$\mathcal{M}[\![\langle p(\vec{t}), \rho\rangle]\!] = \uplus_i \mathcal{M}[\![\langle p_i(\vec{t}), \rho\rangle]\!]$$

where the sole defining clause for p_i is

$$p(\vec{x}) \leftarrow G_i | C_i$$

Proof: The proof is a simple application of the linearity($\uplus$-preserving) properties of various operators. First, note that

$$\begin{aligned}
\mathcal{E}[\![p(\vec{t})]\!] &= \Pi_{ENV} \circ (Id_{ENV} \times^T \mathcal{E}[\![p]\!]) || (\{\!| \mathcal{E}[\![\vec{u}]\!] |\!\}) \circ \langle Id_{ENV}, p\vec{er}p\rangle \\
&= \Pi_{ENV} \circ (Id_{ENV} \times^T [\uplus_i \mathcal{E}[\![p_i]\!]]) || (\{\!| \mathcal{E}[\![\vec{u}]\!] |\!\}) \circ \langle Id_{ENV}, p\vec{er}p\rangle \\
&= \uplus_i [\Pi_{ENV} \circ (Id_{ENV} \times^T \mathcal{E}[\![p_i]\!]) || (\{\!| \mathcal{E}[\![\vec{u}]\!] |\!\}) \circ \langle Id_{ENV}, p\vec{er}p\rangle] \\
&= \uplus_i [\mathcal{E}[\![p_i(\vec{t})]\!]]
\end{aligned}$$

So, we have

$$\begin{aligned}
\mathcal{M}[\![\langle p(\vec{t}), \rho\rangle]\!] &= \mathcal{E}[\![p(\vec{t})]\!] || \mathcal{E}[\![EQ(\rho)]\!] \\
&= [\uplus_i \mathcal{E}[\![p_i(\vec{t})]\!]] || \mathcal{E}[\![EQ(\rho)]\!] \\
&= \uplus_i [\mathcal{E}[\![p_i(\vec{t})]\!] || \mathcal{E}[\![EQ(\rho)]\!]] \qquad \blacksquare
\end{aligned}$$

Corollary 1 *Let Let the defining clauses for p be given by*

$$p(\vec{x}) \leftarrow G_1 | C_1$$

$$\vdots p(\vec{x}) \leftarrow G_m | C_m$$

Consider the configuration $\langle p(\vec{t}), \rho\rangle$. Furthermore, assume that $\mathcal{M}[\![\langle p_i(\vec{t}), \rho\rangle]\!] env_\perp = T$. Then,

$$\mathcal{M}[\![\langle p(\vec{t}), \rho\rangle]\!] = \uplus_i [\mathcal{M}[\![\langle p_i(\vec{t}), \rho\rangle]\!] | i = 2 \ldots n$$

where the sole defining clause for p_i is

$$p(\vec{x}) \leftarrow G_i | C_i$$

Proof: The result follows by above lemma, by observing that for all elements S of $\overline{P_S(ENV)}$, $S \uplus \{T\} = S$. $\blacksquare$

The following lemma is the heart of the proof. Intuitively, this lemma states that, under some restrictions, the replacement of a predicate name by its definition does not change the denotation of a configuration.

Lemma 12 *Let the unique defining clause for p be*

$$p(\vec{x}) \leftarrow G | C$$

where the variables in $\vec{x}$ are distinct. Also, assume that the above clause has distinct names from ρ. Consider the configuration $\langle p(\vec{t}), \rho\rangle$. Then,

$$\mathcal{M}[\![\langle p(\vec{t}), \rho\rangle]\!] = \mathcal{M}[\![\langle G | C, \rho \bigcup \{\vec{x} = \vec{t}\}\rangle]\!]$$

Proof: Recall that the denotation of $p(\vec{t})$ was defined as follows:

$$\mathcal{E}[\![p(\vec{t})]\!] = \Pi_{ENV} \circ (Id_{ENV} \times^T \mathcal{E}[\![p]\!]) || (\{\!\![\mathcal{E}[\![\vec{t}]\!]]\!\!\}) \circ \langle Id_{ENV}, pe\vec{r}p \rangle$$

where $\mathcal{E}[\![p]\!]$ was defined as

$$\mathcal{E}[\![p]\!] = \lambda \vec{a}.\Pi_{\vec{x}} \circ \mathcal{E}[\![G|C]\!] \circ [env_{\perp}[\vec{x} \mapsto \vec{a}]$$

Consider $\mathcal{M}[\![\langle G|C, \rho \bigcup \{\vec{x} = \vec{t}\,\rangle]\!]$. Since, the variables in $G|C$ are all distinct from the variables in ρ, $\mathcal{E}[\![G|C]\!]$ can be rewritten as $Id_{ENV} \times^T \mathcal{E}[\![G|C]\!]$ to indicate that $\mathcal{E}[\![G|C]\!]$ affects only the variables $\vec{x}$, and ENV is the variables that are not any of the $\vec{x}$. So, we have:

$$\begin{aligned} \mathcal{M}[\![\langle G|C, \rho \bigcup \{\vec{x} = \vec{t}\,\rangle]\!] &= \mathcal{E}[\![G|C]\!] || \mathcal{E}[\![EQ(\rho \bigcup \{\vec{x} = \vec{t}\})]\!] \\ &= (Id_{ENV} \times^T \mathcal{E}[\![G|C]\!]) || \mathcal{E}[\![EQ(\rho \bigcup \{\vec{x} = \vec{t}\})]\!] \\ &= (Id_{ENV} \times^T \mathcal{E}[\![G|C]\!]) || \{\vec{x} = \vec{t}] | \mathcal{E}[\![EQ(\rho)]\!] \end{aligned}$$

Since we are interested only in the effect on the variables in ρ, the actual function of interest is

$$\Pi_{ENV} \circ \mathcal{M}[\![\langle G|C, \rho \bigcup \{\vec{x} = \vec{t}\,\rangle]\!]$$

Note that,

$$\Pi_{ENV} \circ (Id_{ENV} \times^T \mathcal{E}[\![G|C]\!]) || \{\vec{x} = \vec{t}\} || EQ(\rho)$$

can be rewritten as

$$\Pi_{ENV} \circ (Id_{ENV} \times^T \mathcal{E}[\![p]\!]) || (\{\!\![\mathcal{E}[\![\vec{t}]\!]]\!\!\}) \circ \langle Id_{ENV}, pe\vec{r}p \rangle$$

becuase the effect of the equation $\{\vec{x} = \vec{t}\}$ is captured by the parallel composition operation $(Id_{ENV} \times^T \mathcal{E}[\![p]\!]) || (\{\!\![\mathcal{E}[\![\vec{t}]\!]]\!\!\})$. Hence, we have the result. ∎

Theorem 1 *Let $conf = \langle C, \rho \rangle$ be an initial configuration. Let $conf_1 \ldots conf_m$ be such that every non-failing computation path of $conf$ passes through one of $conf_1 \longrightarrow conf_m$. Then,*

$$\mathcal{M}[\![conf]\!]env = \uplus_i \mathcal{M}[\![conf_i]\!]$$

Proof: The proof follows by induction on the number of steps of reduction, if we prove the result for one step reductions. That is, if $conf_1 \ldots conf_m$ be all the possible configurations attainable in one step from $conf$, then

$$\mathcal{M}[\![conf]\!]env = \uplus_i \mathcal{M}[\![conf_i]\!]$$

But, this is immediate from the previous two lemmas and the compositionality of the denotational sematntics. ∎

B Computational adequacy

In this section, we prove that the operational semantics actually attains the values predicted by the denotational semantics. Along with the fact that one-step reduction preserves meaning, this means that the results predicted by the operational and denotational semantics match exactly; this is usually called computational adequacy [13]. The difficult constituent of this proof is that one has to construct a reduction sequence from semantic information.

B.1 Relating Denotational and Operational environments

For defining the inclusive predicate relating predicate symbols and partial closure operators, we need to develop notation that relates syntactic and semantic values, syntactic and syntactic environments.

Recall that a notion of the value of a term in a syntactic environemnt was defined. Th definition is reprodcued below. First, we define formally the value of a term in a syntactic environment. Let ρ be a syntactic environment, in reduced form, that is consistent. Consider the following transition system. Let s denote a finite sequence. Let $t = f_i^n(t_1, \ldots t_n)$ be any term. Then, we define $t \uparrow s$ inductively as follows:

- $t \uparrow 0 = f_i^n$
- $t_i \uparrow s = \mathrm{g} \Rightarrow t \uparrow [i|s] = \mathrm{g}$

The following rules "evaluate" an expression of form $t \uparrow s$ in an environment ρ.

1. $< x, \rho > \rightarrow undefined$
   if the alias set of x contains no non-variable terms.

2. $< x, \rho > \rightarrow t$
   if t is in the alias set of x. Note that there may be may be many different terms in the alias set of x. t is arbitrarily chosen from this alias set, by some rule, say lexicographic ordering. (The following lemma essentially states that this seemingly arbitrary choice does not affect the results of the evaluation of $\langle e, \rho \rangle$, in the interesting cases)

3. $\dfrac{< e, \rho > \rightarrow t}{< e \uparrow s, \rho > \rightarrow t \uparrow s}$

Lemma 13 *Let ρ be a consistent syntactic environment. Let $< e, \rho > \rightarrow f_n^i$, where f_n^i is a function symbol. Then, $< e, \rho > \rightarrow f_n^i$ is independent of the choice made in rule 2 of the transition system above.*

The following sequence of definitions are intended to set a up a relationship between syntactic and semantic environments. The following definition relates syntactic expressions and semantic values. Intuitively, $v \preceq (t, \rho)$ means that that t when evaluated in syntactic environment ρ gives a value that is more defined than v. t *DOMINATES* v in ρ, written $v \preceq (t, \rho)$ is defined inductively as follows:

Definition 16 *(Relating terms and semantic values)*

- $v \preceq (x,\rho)$ *if for all finite sequences s,*
  $v[s] \in W_f \Rightarrow \langle x \uparrow [s|0], \rho\rangle \xrightarrow{*} f$

- $v_1 \preceq (t_1,\rho) \wedge \ldots v_n \preceq (t_n,\rho) \Rightarrow f(v_1 \ldots v_n) \preceq (f(t_1,\ldots t_n),\rho)$

We write $\vec{a} \preceq (\vec{t},\rho)$ as shorthand for $a_i \preceq (t_i,\rho), i = 1\ldots n$.

The following definition relates syntactic and semantic environments. It can be viewed as saying that the syntactic environment ρ is more constrained than *env*. ρ dominates *env*, written $env \preceq \rho$, is defined as follows:

Definition 17 *(Relating semantic and syntactic environents)*
$env \preceq \rho \Leftrightarrow$

$$\rho \text{ consistent} \Rightarrow env \neq env_T \wedge (\forall x)\ [env(x) \preceq (x,\rho)]$$

Both the relations defined above possess some monotonicity properites.

Lemma 14 *(Monotonicity properties of $\preceq$)*

- $v' \sqsubseteq v \Rightarrow\ [v \preceq (t,\rho) \Rightarrow v' \preceq (t,\rho)]$

- $env' \sqsubseteq env \Rightarrow\ [env \preceq \rho \Rightarrow env' \preceq \rho]$

The following lemma states that the relations $\preceq$ defined above is *inclusive*([24]). The proof is immediate and is omitted.

Lemma 15 *(Inclusive predicates)*

- *Let* $v = \bigsqcup_i \{v_i\}$. *Then*
$$[(\forall v_i)\ v_i \preceq (t,\rho)] \Rightarrow v \preceq (t,\rho)$$

- *Let* $env = \bigsqcup_i \{env_i\}$. *Then*
$$[(\forall env_i)\ env_i \preceq \rho] \Rightarrow env \preceq \rho$$

The above two definitions can be combined in the natural manner.

Definition 18 $env, a \preceq \rho, t \Leftrightarrow\ a \preceq (t,\rho) \wedge env \preceq \rho$

B.2 Determinate case

First, we assume that there is only one definition per predicate name.

Definition 19 *(Definition of relation $\longrightarrow_s$)*

- $conf_1 \longrightarrow conf_2 \Rightarrow conf_1 \longrightarrow_s conf_2$

- $\langle C,\rho\rangle\ \longrightarrow_s \langle C, C', \rho\rangle$

When $\langle C,\rho\rangle \xrightarrow{*}_s \langle D,\rho'\rangle$, we will sometimes write $\langle C,\rho\rangle \xrightarrow{*}_s \langle U_C \uplus U_e, \rho'\rangle$, to indicate that the atoms in U_C arose by reductions from C. Thus the atoms in U_e arise from the atoms addded in step two of the above definition, the subscript e is intended to indicate "extra".

Lemma 16 $(\langle C,\rho\rangle \longrightarrow_s \langle C,\rho'\rangle) \wedge env \preceq \rho \Rightarrow env \preceq \rho'$

The following lemma is a restricted Church-Rosser like property.

Lemma 17 *Let the set of clauses satisfy the property that there is atmost one definition for each predicate. Let* $\langle\{C_i\},\rho\rangle$ *be a configuration. Let* $conf_1$ *and* $conf_2$ *be two configurations such that*

$$\langle\{C_i\},\rho\rangle \longrightarrow conf_1 \ \wedge \ \langle\{C_i\},\rho\rangle \longrightarrow conf_2$$

Then, there is a configuration $conf$ *such that*

$$conf_1 \longrightarrow conf \ \wedge \ conf_2 \longrightarrow conf$$

Lemma 18 *Let the set of clauses satisfy the property that there is atmost one definition for each predicate. let* $\langle C,\rho\rangle$ *be a configuration. Let* $conf_1$ *and* $conf_2$ *be two configurations such that*

$$\langle C,\rho\rangle \xrightarrow{*}_s conf_1 \ \wedge \ \langle C,\rho\rangle \xrightarrow{*}_s conf_2$$

Then, there is a configuration $conf$ *such that*

$$conf_1 \xrightarrow{*}_s conf \ \wedge \ conf_2 \xrightarrow{*}_s conf$$

Definition 20 *(Definition of inclusive predicate for terms)*
Let $g \in [(ENV \times^T \Pi_n^T V) \xrightarrow{c} (ENV \times^T \Pi_n^T V) \times 2)]$. $\vec{t} \preceq g$ *if , we have*

$$env, \vec{a} \preceq \rho, \vec{t} \wedge \rho \neq error \Rightarrow r \preceq \rho, t$$

where $g\langle env, \vec{a}\rangle = \langle r, term\rangle$

The following definition is the definition of the relation $g \preceq C$, between sequences of atoms C and closure operators g on ENV. The following definition can be intuitively saying that every finite portion of the answer predicted by the denotational semantics is attained by the operatonal semantics after a finite number of steps. In particular, the semantics handles failed computations too.

Definition 21 *(Definition of inclusive predicate)*
Let $g \in [(ENV) \xrightarrow{c} (ENV) \times 2)]$. $g \preceq C$ *if the following holds. Let,*

- $\langle C, \star\rangle \xrightarrow{*}_s \langle U_C \bigcup U_e, \rho\rangle$
- $env \preceq \rho$

Let $g(env) = \langle env', term\rangle$. *Then,* $(\forall env_f \sqsubseteq env')$ $(\exists\langle U'_C \uplus U'_e, \rho'\rangle)$ *such that*

- $\langle U_C \uplus U_e, \rho\rangle \xrightarrow{*} \langle U'_C \uplus U'_e, \rho'\rangle)$
- $env_f \preceq \rho'$
- $term = T \Rightarrow U'_C = \phi$

Suppose that g_1 and g_2 are partial two closure operators that correspond to the imposition of two constraints given as sequences C_1 and C_2. Suppose that we know how to construct reduction sequences corresponding to C_1 and C_2 individually. Then, from the definition of the parallel composition g_1 and g_2, we can construct an interleaved reduction sequence of C_1 and C_2 corresponding to the computing the iterates of $(g_1 \circ g_2)$. In other words, the special form of the fixed point iteration corresponding to the parallel composition operation provides guidance about how to construct the interleaved reduction sequence. The proof of the following lemma formalizes this intuition.

Lemma 19 *Let* $g, h \in [(ENV) \xrightarrow{c} (ENV) \times \mathbf{2})]$. *Then,*

$$g \preceq C \wedge h \preceq D \Rightarrow g||h \preceq C, D$$

Proof: Let

- $g||h\ env = \langle env', term\rangle$
- $\langle C, D, \star\rangle \xrightarrow{*}_s \langle U_C \uplus U_D \uplus U_e, \rho\rangle$
- $env \preceq \rho$

Let $\langle env_i, term_i\rangle = (g \circ h)^i\ env$. We prove by induction on i that

$$(\forall env_f \sqsubseteq env_i)\ (\exists \langle U'_C \uplus U'_D \uplus U_e, \rho_{res}\rangle)$$

such that

- $\langle U_C \uplus U_D \uplus U_e, \rho\rangle \xrightarrow{*} \langle U'_C \uplus U'_D \uplus U_e, \rho_{res}\rangle$
- $env_f \preceq \rho_{res}$.
- $term_i = T \Rightarrow [U'_C \uplus U'_D = \phi]$

Base: $(i = 0)$
In this case, $env_f \sqsubseteq env$ and the configuration $\langle U_C \uplus U_D \uplus U_e, \rho\rangle$ satisfies required properties.
Induction: (assume result for i)

From the continuity of all functions involved, we deduce the existence of finite environments env_1 and env_2 such that,

- $\langle env_f, term_{(i+1)}\rangle \sqsubseteq g\ env_1$
- $\langle env_1, term_{(i+1)}\rangle \sqsubseteq h\ env_2$
- $\langle env_2, \perp\rangle \sqsubseteq (g \circ h)^i env$

We can assume the same second coordinates for both $g\ env_1$ and $h\ env_2$, because we know that the second coordinate of $g||h$ applied to env is the greatest lower bound of the second coordinates of g and h applied to env. This is the semantic way of capturing the intuitive idea that $g||h$ terminates if and only if both h and g terminate. From induction hypothesis, $(\exists \langle U''_C \uplus U''_D \uplus U_e, \rho_2\rangle)$ such that

- $\langle U_C \uplus U_D \uplus U_e, \rho\rangle \xrightarrow{*} \langle U''_C \uplus U''_D \uplus U_e, \rho_2\rangle$
- $env_2 \preceq \rho_2$.

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

Since $\langle U_D, \rho\rangle \xrightarrow{*}_s \langle U_C'' \uplus U_D'' \uplus U_e, \rho_2\rangle$ and $h \preceq D$, $(\exists \langle U_C'' \uplus U_D' \uplus U_e, \rho_2\rangle)$ such that

- $\langle U_C'' \uplus U_D'' \uplus U_e, \rho_2\rangle \xrightarrow{*} \langle U_C'' \uplus U_D' \uplus U_e, \rho_1\rangle$
- $env_1 \preceq \rho_1$
- $term_i = T \Rightarrow U_D' = \phi$

Since $\langle U_C, \rho\rangle \xrightarrow{*}_s \langle U_C'' \uplus U_D' \uplus U_e, \rho_1\rangle$ and $g \preceq C$, $(\exists \langle U_C' \uplus U_D' \uplus U_e, \rho_{res}\rangle)$ such that

- $\langle U_C'' \uplus U_D' \uplus U_e, \rho_1\rangle \xrightarrow{*} \langle U_C' \uplus U_D' \uplus U_e, \rho_{res}\rangle$
- $env_f \preceq \rho_{res}$
- $term_i = T \Rightarrow U_C' = \phi$

Hence,we have the result. ∎

Corollary 2 $\vec{t} \preceq \mathcal{E}[\![\vec{t}]\!]$

Proof: Structural induction on terms. The result is immediate for variables. The inductive step for sequences of terms uses the above result.

Corollary 3 *Let* $g_1, g_2 \in [(ENV) \xrightarrow{c} (ENV) \times 2\!\!\!/)]$. *Then,*

$$g_2 \preceq C \wedge g_2 \preceq C \Rightarrow g_1 || g_2 \preceq C$$

Proof: Proof identical to proof of lemma. ∎

Corollary 4 $\mathcal{E}[\![t_1 = t_2]\!] \preceq \{t_1 = t_2\}$

Lemma 20 $g \preceq C \Rightarrow \mathcal{E}[\![G]\!] | g \preceq G | C$

Proof: Note that the guard predicates satisfy the following "adequacy property". Let $env \preceq \rho$. Then,

- $\mathcal{E}[\![G]\!]\ env = tt \Rightarrow (G, \rho) = true$
- $\mathcal{E}[\![G]\!]\ env = ff \Rightarrow (G, \rho) = false$

In case 1, result follows from assumption $g \preceq C$. In case 2, $(\mathcal{E}[\![G]\!] | g)\ env = env_T$. Operationally, guard G evaluates to false and the execution of $G|C$ in ρ fails. ∎

Recall that we defined the meanings of predicate symbols as elements of $[(\Pi_n^T V) \xrightarrow{c} (\Pi_n^T V) \times 2\!\!\!/)]$. So,we need to develop some notation to relate elements of $[(\Pi_n^T V) \xrightarrow{c} (\Pi_n^T V) \times 2\!\!\!/)]$ and predicate symbols. The motivations for these definitions are quite the same as before.

Definition 22 *(Definition of inclusive predicate)*
Let $h \in [(\Pi_n^T V) \xrightarrow{c} (\Pi_n^T V) \times 2)]$. $h \preceq p$ *if the following holds. Let*

- $\langle p(\vec{t}), \star\rangle \xrightarrow{*}_s \langle U^p \uplus U, \rho\rangle$
- $\vec{a} \preceq (\vec{t}, \rho)$

If, $h\vec{a}_f = \langle res, term\rangle$, $(\forall \langle \vec{b}_f, term\rangle \sqsubseteq res)$ $(\exists \langle U'_p \uplus U', \rho'\rangle)$ *such that*

- $\langle U_p \uplus U, \rho\rangle \xrightarrow{*} \langle U'_p \uplus U', \rho'\rangle)$
- $\vec{b}_f \preceq (\vec{t}, \rho')$
- $term = T \Rightarrow U' = \phi$

The following lemma captures the notion of alpha-conversion, in the semantics.

Lemma 21 *(Renaming lemma)*
Let $\lambda : Y_1 \rightarrow Y_2$ *be a bijection between two disjoint sets of variables* Y_1, Y_2. *Let* $C \preceq g$, *where* $Var(C) \subseteq Y_1$. *Then,* $C_\lambda \preceq g_\lambda$, *where*

- C_λ *denotes the result of simultaneous substitution of the variables* $x \in Y_1$ *by* $\lambda(x) \in Y_2$.
- $g_\lambda = \Lambda \circ g \circ \Lambda$, *where*

$$\Lambda(env)(x) = \begin{cases} env(\lambda(x)) & x \in Y_1 \\ env(\lambda^{(-1)}(x)) & x \in Y_2 \\ env(x) & x \notin Y_1 \bigcup Y_2 \end{cases}$$

Proof: Note that $C \preceq g \Rightarrow (x \notin Y_1 \Rightarrow [\Pi_1 \circ g(env)](x) = env(x))$. Result now follows from definitions. ∎

The following is the case of the structural induction that enables us to deduce the desired properties for the predicate name p given that the properties hold for the clause body defining p.

Lemma 22 $g \preceq \{G|C\} \Rightarrow h \preceq p$ *where*

- *The defining clause for* p *is* $p(\vec{x}) \leftarrow G|C$.
- $h = \Pi_{\vec{x}} \circ \mathcal{E}[\![G|C]\!] \circ env_\perp[\vec{x} \mapsto \vec{v}]$

Proof: Let

- $\langle p(\vec{t}), \star\rangle \xrightarrow{*}_s \langle U_p \uplus U, \rho\rangle$
- $\vec{a} \preceq (\vec{t}, \rho)$

Let $\vec{y}$ be variables not found in ρ or $U_p \uplus U$. Note that

$$\langle p(\vec{t}), \star\rangle \longrightarrow \langle G_y^x | C_y^x, \star\rangle$$

where $G_y^x | C_y^x$ is supposed to indicate a freshly renamed clause with variables $\vec{y}$ for $\vec{x}$. From 18, we deduce the existence of a configuration $conf$ such that

$$\langle G_y^x | C_y^x, \star\rangle \xrightarrow{*}_s conf \wedge \langle U, \rho\rangle \xrightarrow{*} conf$$

From lemma 21, $g^x_y \preceq G^x_y|C^x_y$. Furthermore, $\vec{a} \preceq (\vec{t}, \rho) \Rightarrow \vec{a} \preceq (\vec{t}, \rho_{conf})$. Thus, we deduce $\vec{a} \preceq (\vec{y}, \rho_{conf})$. Result follows from hypothesis $g \preceq \{G|C\}$. ∎

The following lemma completes the cycle of structural induction proofs. The lemma proves that the goal atoms of form $p(\vec{t})$ inherit desired properties from p and $\vec{t}$.

Lemma 23

$$h \preceq p \wedge g \preceq \vec{t} \Rightarrow h_g \preceq p(\vec{t})$$

where $h_g = \Pi_{ENV} \circ (Id_{ENV} \times^T h)||g$.

Proof: Note that we have,

- $h \preceq p \Rightarrow Id_{ENV} \times^T h \preceq p(\vec{t})$
- $g \preceq \vec{t} \Rightarrow g \preceq p(\vec{t})$

Using corollary 3, $p(\vec{t}) \preceq Id_{ENV} \times^T h||g$. ∎

B.3 Determinisation

The motivation of this subsection is to set up tools to designate the possible execution sequences of a program. Informally, this is done by associating with each predicate name a sequence of integers that indicate the definition chosen at a reduction step. The sequence of integers can be thought of as an oracle that identifies the choice to be made. For example, let a predicate name p has 5 definitions. Then, a sequence of integers starting with 3 indicates that the third definition is chosen at the first call of p. However, note that we need more structure. Continuing the example sketched above, let the third definition of p in the example above have the form

$$p(\vec{t}) \leftarrow G|q_1(\vec{t_1}), q_2(\vec{t_2})$$

Also assume that each of q_1 and q_2 have 3 definitions each. So, the information needed to determine an execution sequence of p completely should contain the choices to be made for q_1 and q_2. For example, a tuple of the form $\langle 3, 1, 2\rangle$ identifies the choices to be made when reducing an atom with head p: this information can be read as "Use the third definition for p, the first definition for q_1 and the second definition for q_2 ". Note that q_1 or q_2 might be p. Thus, we need definitions that can handle possible infinite reduction sequences. The following definitions should bne construed as one way of stating everything formally.

Let N^ω be the set of all finite and infinite sequences over the natural numbers N. Define R inductively as follows:

$$\begin{array}{rcl} N^\omega & \subseteq & R \\ (\forall n)\,[s_1 \ldots s_n \in R & \Rightarrow & \langle s_1 \ldots s_n\rangle \in R] \end{array}$$

Let $L = \Pi_p P(R)$, where p are the predicate names in the program. Thus, L is a product of copies of the powerset of R, the copies indexed by the predicate names p. Note that L is a complete lattice under pointwise subset inclusion. With each predicate symbol p, we will associate a subset S_p of L, that will denote the determinisations of p. The sets $\{S_p\}$ are constructed as the maximal fixpoint of a monotone function F on L. Let $\{S_p\}$ be an element of L. Let the ith definition of p be

$$p(\vec{t}) \leftarrow G_i^p | q_{i1} \dots q_{iu_i}$$

Then, $F(\{S_p\}) = \{T_p\}$ is defined as follows:

$$\langle i, s_{i1} \dots s_{iu_i} \rangle \in T_p \Rightarrow (s_{ij} \in S_{q_{ij}}, 1 \leq j \leq u_i)$$

It is easy to check that F is monotone. Since F is a monotone function on a complete lattice, F has a maximum fixed point. Let $\{Det(p)\}$ be the maximum fixed point of F.

Now, the definition of determinisation can be extended to sequences of atoms. A determinisation of $q_1(.), \dots q_n(.)$ is an n-tuple $\langle c_1 \dots c_n \rangle$, where $c_1 \in Det(q_i)$. Similarly a determinisation of $G|C$ is just a determinisation of C.

In the sequel, we will not explicitly refer to the structure of the sequences of the determinisation of a program. Instead, we will identify the sequences with the (possibly infinite) deterministic program that they encode.

B.4 Full proof

Definition 23 *Let* $H \in \overline{P([\![(\Pi_n^T V) \xrightarrow{c} (\Pi_n^T V) \times \mathbf{2})]\!])}$. *Then,* $H \preceq p$ *if*

$$(\forall p_d \in Det(p))\,(\exists h)\,[H \uplus \{\!|\; h \;|\!\} = H \wedge \; h \preceq p_d]$$

Definition 24 *For* $F \in \overline{P([\![(ENV \times^T \Pi_n^T V) \xrightarrow{c} (ENV \times^T \Pi_n^T V) \times \mathbf{2})]\!])}$, *we define* $F \preceq p(\vec{t})$ *if*

$$(\forall p^d \in Det(p))\,(\exists f)\,[F \uplus \{\!|\; f \;|\!\} = F \wedge f \preceq p_d(\vec{t}]$$

Definition 25 *Let* $G \in \overline{P([\![(ENV) \xrightarrow{c} (ENV) \times \mathbf{2})]\!])}$. *Then,* $G \preceq C$ *if*

$$(\forall c^d \in Det(C))\,(\exists g)\,[G \uplus \{\!|\; g \;|\!\} = G \wedge \; g \preceq c^d]$$

Lemma 24 *All the predicates defined above are inclusive.*

Lemma 25 *(Cases of structural induction)*

1. $G_1 \preceq C_1 \wedge G_2 \preceq C_2 \Rightarrow G_1 || G_2 \preceq C_1, C_2$

2. $G \preceq C \Rightarrow \mathcal{E}[\![Gu]\!]|C \preceq Gu|C$

3. *Let the defining clauses for* p *be* $p(\vec{u_1}) \leftarrow Gu_1|C_1 \dots p(\vec{u_n}) \leftarrow Gu_n|C_n$. *Define new predicate names* p_i, *for* $i = 1 \dots n$, *by the unique clause* $p_i(\vec{u_i}) \leftarrow Gu_i|C_i$. *Then,*

$$(\forall i = 1 \dots n)\,(H_i \preceq p_i) \Rightarrow \uplus_i H_i \preceq p$$

4. *Let* $G_1 \preceq Gu|C\}$ *and the unique defining clause for* p *be* $p(\vec{x} \leftarrow Gu|C$. *Then,* $H \preceq p$ *where* $H = \Pi_{\vec{x}} \circ \mathcal{E}[\![G|C]\!] \circ env_\perp[\vec{x} \mapsto \vec{v}]$

5.

$$H \preceq p \wedge g \preceq \vec{t} \Rightarrow H_g \preceq p(\vec{t})$$

where $H_g = (Id_{ENV} \times^T H) || \{\!| g |\!\}$.

Proof: The proofs reduce the cases of structural induction to the corresponding cases of the structural induction for the determiante case.

1. Every element of $Det(C_1, C_2)$ has the form $\langle c_1^d, c_2^d \rangle$ where $c_i^d \in Det(C_i)$. From hypothesis $F_1 \preceq C_1 \wedge F_2 \preceq C_2$, there are g_1, g_2 such that

$$g_1 \preceq c_1^d \wedge G_1 \uplus \{\!| g_1 |\!\} = G_1$$

and

$$g_2 \preceq c_2^d \wedge G_1 \uplus \{\!| g_1 |\!\} = G_1$$

From lemma 19, we deduce that $g_1 || g_2 \preceq c_1^d, c_2^d$. Furthermore,

$$\begin{aligned} F||G &= F \uplus \{\!| f |\!\} || G \\ &= F||G \uplus \{\!| f |\!\} || G \uplus \{\!| g |\!\} \\ &= F||G \uplus \{\!| f |\!\} || G \uplus \{\!| f |\!\} || \{\!| g |\!\} \\ &= F \uplus f || G \uplus \{\!| f||g |\!\} \\ &= F||G \uplus \{\!| f||g |\!\} \end{aligned}$$

2. Every element of $Det(Gu|C)$ is of form $Gu|c^d$ where $c^d \in Det(C)$. From hypothesis on C there is a g such that

$$g \preceq c^d \wedge G \uplus \{\!| g |\!\} = G$$

From lemma 20 $\mathcal{E}[\![Gu]\!]|g \preceq Gu|c$. Furthermore,

$$\begin{aligned} \mathcal{E}[\![Gu]\!]|G = & \ \mathcal{E}[\![Gu]\!]|(G \uplus \{\!| g |\!\}) \\ = & \ \mathcal{E}[\![Gu]\!]|G \uplus \mathcal{E}[\![Gu]\!]|\{\!| g |\!\}) \\ = & \ \mathcal{E}[\![Gu]\!]|G \uplus \{\!| \mathcal{E}[\![Gu]\!]|g |\!\} \end{aligned}$$

3. Given an element $p^d \in Det(p)$, look at the first element of p^d. If the first element is i, the rest of p^d induces a determinisation p_i^d, of p_i. From assumption, $H_i \preceq p_i$, there exists h such that

$$H_i \uplus h_i = H \wedge h_i \preceq p_i^d$$

Hence, we have

$$H \uplus h_i = H \wedge h_i \preceq p^d$$

4. Every element $p^d \in Det(p)$ has the form $Gu|c^d$. where $c^d \in Det(C)$. From hypothesis, $G \preceq \{Gu|C\}$, there exists g such that

$$g \preceq \{Gu|c^d\} \wedge G \uplus \{\!| g |\!\} = G$$

From lemma 22, $h \preceq p^d$ where

$$h = \Pi_{\vec{x}} \circ g \circ env_\perp[\vec{x} \mapsto \vec{v}]$$

$$\begin{aligned} H \uplus \{\!\!\{\ h\ \}\!\!\} &= H \uplus \{\!\!\{\ \Pi_{\vec{x}} \circ g \circ env_\perp[\vec{x} \mapsto \vec{v}]\ \}\!\!\} \\ &= \Pi_{\vec{x}} \circ (G \uplus \{\!\!\{\ g\ \}\!\!\}) \circ env_\perp[\vec{x} \mapsto \vec{v}] \\ &= \Pi_{\vec{x}} \circ G \circ env_\perp[\vec{x} \mapsto \vec{v}] \\ &= H \end{aligned}$$

5. Every element $p^d(\vec{t}) \in Det(p(\vec{t}))$ is determined by an element $p^d \in Det(p)$. From assumption $H \preceq p$, there exists h such that
$$h \preceq p^d \wedge H \uplus h = h$$
Consider $h_g = Id_{ENV} \times^T h || g$. From lemma 23, $h_g \preceq p^d$. Furthermore,
$$\begin{aligned} H_g \uplus \{\!\!\{\ h_g\ \}\!\!\} &= (Id_{ENV} \times^T H) || \{\!\!\{\ g\ \}\!\!\} \uplus \{\!\!\{\ (Id_{ENV} \times^T h) || g\ \}\!\!\} \\ &= (Id_{ENV} \times^T H) || \{\!\!\{\ g\ \}\!\!\} \uplus (Id_{ENV} \times^T \{\!\!\{\ h\ \}\!\!\}) || \{\!\!\{\ g\ \}\!\!\} \\ &= (Id_{ENV} \times^T (H \uplus \{\!\!\{\ h\ \}\!\!\})) || \{\!\!\{\ g\ \}\!\!\} \\ &= (Id_{ENV} \times^T H) || \{\!\!\{\ g\ \}\!\!\} \\ &= H_g \quad \blacksquare \end{aligned}$$

Theorem 2 $\mathcal{E}[\![p]\!] \preceq p$

Proof: The proof is by induction on the order of definitions of predicates. All cases of induction except the case of definitions by recursive are carried out in the lemma 25. We give the proof for the recursive case below.

Let the defining clauses for p be given by

$$\begin{aligned} p(\vec{x}) &\leftarrow Gu_1|C_1 \\ &\vdots \\ p(\vec{x}) &\leftarrow Gu_m|C_m \end{aligned}$$

Recall that $\tau_i(H)$ was defined as follows:

$$\mathcal{E}[\![\tau_i(f)]\!] = \Pi_{(2\ldots n+1)} \circ (\mathcal{E}[\![G_i|C_i]\!] \circ [env_\perp[\vec{x} \mapsto \vec{a}]])$$

Also, we defined $\tau(H) = \uplus_i \tau_i(H)$.

We prove by fixpoint induction that $H \preceq p \Rightarrow \tau H \preceq p$. From lemma 25 above, it suffices to prove that $H \preceq p \Rightarrow \tau_i H \preceq p_i$, where the sole defining clause is the ith clause of p.

- The ith defining clause for p contains no reference to p. In this case, proof does not require the fixpoint induction hypothesis, and follows from the induction hypothesis on $Gu_i|C_i$ and from teh relevant cases of lemma 25.

- The ith defining clause for p contains references to p. Call this p_i. Proof is by using the relevant cases of lemma 25, using fixpoint induction hypothesis, to deduce that $\tau_i(H) \preceq p_i$

Note that $\mathcal{E}[\![p]\!] = \bigsqcup_i \tau^i(\perp)$. Using the inclusivity of the predicate from lemma 24, we get $\mathcal{E}[\![p]\!] \preceq p$.
$\blacksquare$

C Full-abstraction

The aim of this section is to use the proofs relating the operational and denotational semantics coincide in their view of programs.

For this, we first give a translation of primitive observations into finite elements of ENV. The translation function $Trans$ maps elements of OBS to finite elements of $\overline{P_S(V)}$. For the purposes of this definition, we assume that the elements of OBS are in disjunctive normal form, i.e as disjunctions of conjunctions.

1. $Trans(t_1 \sqsubseteq \rho(x_1) \wedge \ldots t_n \sqsubseteq \rho(x_n)) = env_\perp[x_1 \mapsto t, \ldots x_n \mapsto t_n]$,

2. $Trans(p_1 \vee p_2 \ldots p_n) = \uplus_i Trans(p_i)$

The following lemma establishes a tight connection between the tests passed by a program and the denotation of the program.

Lemma 26 $\langle p(\vec{t}), \phi\rangle \vdash s \Leftrightarrow Trans(s) \sqsubseteq \mathcal{E}[\![p(\vec{t})]\!]\langle env_\perp, \perp\rangle$

Proof: Let $Trans(s) \sqsubseteq \mathcal{E}[\![p(\vec{t})]\!] env_\perp$. Consider any valid execution sequence from the configuration $\langle p(\vec{t}), \phi\rangle$. The execution sequence induces a determinisation $p^d \in Det(p(\vec{t}))$. From lemma 2, we have g such that $g \uplus \mathcal{E}[\![p(\vec{t})]\!] = \mathcal{E}[\![p(\vec{t})]\!] \wedge g \preceq p^d$. Let $g\ env_\perp = r$. Let $s^d = \langle env_f, term\rangle \in Trans(s)$ be such that $s^d \sqsubseteq r$. Note that s^d corresponds to a disjunct in s. From definition 21 of $g \preceq p^d$ in Appendix B, there is a configuration $conf = \langle U, \rho\rangle$ such that :

- The execution sequence from $\langle p(\vec{t}), \phi\rangle$ corresponding to p^d reduces in finitely many steps to $conf$
- $env_f \preceq \rho \wedge [term = T \Rightarrow U = \phi]$

Thus, the execution sequence p^d passes test s^d and hence the test s. Since the proof is true for all execution sequences, we deduce that $\langle p(\vec{t}), \phi\rangle \vdash s$.

Let $\langle p(\vec{t}), \phi\rangle \vdash s$. From definitions, every execution path passes test s. Since every predicate symbol has only finitely many definitions, König's lemma proves the existence of finitely many configurations $conf_i, i = 1 \ldots n$ such that

- Every valid execution sequence passes through one of the $conf_i$
- Forall i, $conf_i$ passes test s

From lemma 1,

$$\mathcal{E}[\![\langle p(\vec{t}), \phi\rangle]\!] env_\perp = \uplus \mathcal{E}[\![conf_i]\!] env_\perp$$

Thus,

$$conf_i \vdash s \quad \Rightarrow \quad Trans(s) \sqsubseteq \mathcal{E}[\![conf_i]\!] env_\perp$$

$$\Rightarrow Trans(s) \sqsubseteq \uplus_i \mathcal{E}[\![conf_i]\!] env_\perp \quad \blacksquare$$

Theorem 3 *(Full abstraction)* $\mathcal{E}[\![p]\!] = \mathcal{E}[\![q]\!] \Leftrightarrow$ *p and q are operationally indistinguishable.*

Proof: First, we prove the forward implication. This is called adequacy. Let $\langle C[], \rho\rangle$ be any context. Then, from the compositionality of the semantics, we deduce that $\mathcal{E}[\![\langle C[p], \rho\rangle]\!] = \mathcal{E}[\![\langle C[q], \rho\rangle]\!]$. In particular,

$$\mathcal{E}[\![\langle C[p], \rho\rangle]\!] env_\perp = \mathcal{E}[\![\langle C[q], \rho\rangle]\!] env_\perp$$

Let s be any finite test. Then

$$\begin{aligned} \langle C[p], \rho\rangle \text{ } passes \text{ } s &\Leftrightarrow Trans(s) \sqsubseteq \mathcal{E}[\![\langle C[p], \rho\rangle]\!] env_\perp \\ &\Leftrightarrow Trans(s) \sqsubseteq \mathcal{E}[\![\langle C[q], \rho\rangle]\!] env_\perp \\ &\Leftrightarrow \langle C[q], \rho\rangle \text{ } passes \text{ } s \end{aligned}$$

Next, we prove the reverse implication. Let $\mathcal{E}[\![p]\!] \neq \mathcal{E}[\![q]\!]$. Then, we have either $\mathcal{E}[\![p]\!] \not\sqsubseteq \mathcal{E}[\![q]\!] \vee \mathcal{E}[\![q]\!] \not\sqsubseteq \mathcal{E}[\![p]\!]$. Assume that $\mathcal{E}[\![p]\!] \not\sqsubseteq \mathcal{E}[\![q]\!]$. Then, there exists a finite $x \in V$ such that $\mathcal{E}[\![p]\!](x) \not\sqsubseteq \mathcal{E}[\![q]\!](x)$. Thus, there is a finite element s of $\overline{P_S(\Pi_n^\top V)}$ such that $s \sqsubseteq \mathcal{E}[\![p]\!](x) \wedge s \not\sqsubseteq \mathcal{E}[\![q]\!]x$. Let t_x be the sequence of finite terms coding x, and let $test_s$ be the finite test corresponding to s. Then, $s \sqsubseteq \mathcal{E}[\![p]\!]x$ implies that $\langle p(t_x), \phi\rangle$ passes $Test_s$. We claim that $\langle p(t_x), \phi\rangle$ does not pass $Test_s$. For, if it did, we have $Trans(Test_s) = s \sqsubseteq \mathcal{E}[\![q(t_x)]\!]$. ∎

D Relating $\overline{P()}$ and $\overline{P_S()}$

The powerdomain construction used in this paper and $\overline{P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])}$ are closely related. The main technical result of this section defines a continuous surjection r from $\overline{P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])}$ onto $\overline{P([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])}$ that preserves all operations of interest. In particular, it preserves $\uplus$, $||$. Only sketches of proofs are presented.

Let $B([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])$ be the basis of $[(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})]$. The basis of the Smyth powerdomain of $[(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})]$, denoted $P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])$, is the finite powerset of the basis elements, denoted $P_{fin}(B([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})]))$. The ordering relation is defined as

$$\{f_1 \ldots f_n\} \sqsubseteq \{g_1 \ldots g_m\} \Rightarrow (\forall 1 \leq j \leq m)\ (\exists 1 \leq i \leq n)\ [f_i \sqsubseteq g_j]$$

The Smyth powerdomain [23], $\overline{P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])}$ is the ideal completion of $P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])$, i.e we have

- The elements are downward closed, directed subsets of $P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])$.
- The ordering relation is $\subseteq$.

Note that the $\overline{P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])}$ can be made into a continuous algebra [23] with a union operation $\uplus$, defined as follows:

$$S_1 \uplus S_2 = \{s_1 \bigcup s_2 | s_1 \in S_1, s_2 \in S_2\}$$

Note that the operation $\uplus$is idempotent, commutative and associative.

Define an application function *App* from $\overline{P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])} \times D$ to $\overline{P_S(D)}$ as follows. We first define *App* on $P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])$, the basis elements.

$$App(\{f_1 \cdots f_n\}, x) = \uplus_i \{f_i(x) | f_i(x) \downarrow$$

Note that $App(\{f_1 \ldots f_n\}, x)$ is undefined if and only if $x \notin \bigcup_i Dom(f_i)$.

Lemma 27 *App is a monotone function from* $P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})]) \times B(D)$ *to* $B(D)$.

So, *App* can be extended uniquely to a continuous function from the domain $\overline{P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])} \times D$ to $\overline{P_S(D)}$.

Notice however, that the ordering is not "extensional", with respect to the application operation. i.e. we may have two sets of functions $\{f_1 \ldots f_n\}$ and $\{g_1 \ldots g_m\}$, such that

$$(\forall x)\ [App(\{f_1 \ldots f_n\}, x) = App(\{g_1 \ldots g_m\}, x)]$$

From the universal properties of the Smyth powerdomain [23] [4], every continuous function from $[(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})]$ into a continuous algebra [4] extends uniquely to a continuous function preserving $\uplus$from the Smyth-powerdomain of $[(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})]$.

There is a singleton embedding function $\{\!|\ |\!\}$ that injects elements of $[(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})]$ into $\overline{P_S([(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})])}$. $\{\!|\ |\!\}$ is defined first on the basis of $[(D) \stackrel{c}{\rightharpoonup} (D) \times \mathbf{2})]$

$$\{\!|\ f\ |\!\} = \{f\}$$

It is easy to check that $\{\!|\ |\!\}$ is monotone. Hence, it can be uniquely extended to a continuous function $\{\!|\ |\!\}$, defined on the whole of $\overline{P_S([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$.

Furthermore, a parallel composition operation can be defined. The following definition is on the basis elements $P_S([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])$.

$$\{f_1 \dots f_n\} || \{g_1 \dots g_m\} = \uplus\{f_i || g_j | i, j\}$$

Note that $||$ as defined above is just the bilinear (preserving $\uplus$in both arguments) extension of $||$ as defined on $[(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})]$. Monotonicity of $||$ as defined above follows from the monotonicity of $||$ on $[(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})]$, and the monotonicity of $\uplus$. So, it can be extended uniquely to a continuous function. The $||$ operation on $\overline{P_S([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$ inherits the desirable properties from the $||$ operation on $[(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})]$.

Lemma 28 *$||$ is commutative and associative.*

The definition of the operation of parallel composition with sharing, requires a notion of $F \times^T Id_{D_2}$, $F \in \overline{P_S([(D_2) \stackrel{c}{\rightarrow} (D_2) \times \mathbf{2})])}$. We define a continuous function $\times^T Id_{D_2}$ with domain $\overline{P_S([(D_1) \stackrel{c}{\rightarrow} (D_1) \times \mathbf{2})])}$ and range $\overline{P_S([(D_1 \times^T D_2) \stackrel{c}{\rightarrow} (D_1 \times^T D_2) \times \mathbf{2})])}$. This is written postfix for readability. As usual, we define $s \times^T Id_{D_2}$, for $s \in P_S([(D_1) \stackrel{c}{\rightarrow} (D_1) \times \mathbf{2})])$. Let $s = \{f_1 \dots f_n\}$.

$$s \times^T Id_{D_2} = (f_1 \times^T Id_{D_2}) \uplus \dots (f_n \times^T Id_{D_2})$$

It is easy to check that the above definition defines a monotone function from $P_S([(D_2) \stackrel{c}{\rightarrow} (D_2) \times \mathbf{2})])$ to $\overline{P_S([(D_1 \times^T D_2) \stackrel{c}{\rightarrow} (D_1 \times^T D_2) \times \mathbf{2})])}$. So, it can be extended uniquely to a continuous function from $\overline{P_S([(D_1) \stackrel{c}{\rightarrow} (D_1) \times \mathbf{2})])}$ to $\overline{P_S([(D_1 \times^T D_2) \stackrel{c}{\rightarrow} (D_1 \times^T D_2) \times \mathbf{2})])}$.

We can define a continuous operator $v_f|$ on $\overline{P_S([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$, where v_f is a finite element of D. $v_f|$ is the unique $\uplus$preserving continuous extension of the function $v_f|$ on $[(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})]$. Unwinding the definitions, we get the definition of $v_f|$ on the finite elements of $\overline{P_S([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$ as:

$$v_f | \{f_1 \dots f_n\} = \uplus_i \{v_f | f_i\}$$

Note that we can define a continuous $\uplus$ preserving surjection r from the Smyth powerdomain onto the new powerdomain. On the basis of the Smyth powerdomain $P_S([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])$, r is defined as

$$r(\{f_1 \dots f_n\}) = \{f_1 \dots f_n\} \downarrow$$

where $\{f_1 \dots f_n\} \downarrow$ indicates the downward closure of $\{f_1 \dots f_n\}$ in the preorder $P([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])$. It is easy to check monotonicity and preservation of $\uplus$. So, r can be uniquely extended to a $\uplus$preserving, continuous function from $\overline{P_S([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$ to $\overline{P([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$.

Lemma 29 *(Coherence properties of powerdomains)*

1. *Let $f \in [(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})]$. Then,*

$$r(\{\!|\ f\ |\!\}) = \{\!|\ f\ |\!\}$$

2. *Let $S \in \overline{P_S([(D) \stackrel{c}{\rightarrow} (D) \times \mathbf{2})])}$, $s \in \overline{P_S(D)} \wedge x \in D$. Then,*

$$s \sqsubseteq App(S, x) \Leftrightarrow s \sqsubseteq App(r(S), x)$$

3. *Let* $S_1, S_2 \in \overline{P_S([(D) \xrightarrow{c} (D) \times 2)])}$. *Then*

$$r(S_1 || S_2) = r(S_1) || r(S_2)$$

4. *Let* $S_i \in \overline{P_S([(D_0 \times D_i) \xrightarrow{c} (D_0 \times D_i) \times 2)])}, i = 1, 2$. *Then*

$$r(||_{D_0}\langle S_1, S_2\rangle) = ||_{D_0}\langle r(S_1), r(S_2)\rangle$$

5. *Let* $v_f \in B(D)$. *Then*

$$r(v_f | F) = v_f | r(F)$$

Proof: It is easy to check the above for finite elements of the Smyth-powerdomain construction. The result for arbitrary elements follows from the continuity of all functions involved. ∎

Furthermore, the semantic definitions in the section on denotational semantics did not use any special properties of the new powerdomain. The definitions only used the functions that were used to compose definitions: for example, $||$, $v_f|$ etc. Thus, teh semantics could have equally well been defined using the Smyth powerdomain construction instead of the new powerdomain. This is possible because all the operators that we have used to define the semantic function are defined for the Smyth powerdomain construction also. In fact, we have a very tight correspondence between the denotations so derived. The following lemma brings out this connection. Note that there are two ways of getting the denotation of a predicate sympbol or term in the new powerdomain.

- Define the denotation directly in the new powerdomain. This was done in the section on the denotational semantics.
- Define the denotation in the Smyth powerdomain, as alluded to above. Note that the function r defined earlier, gives us away of going from the Smyth powerdomain to the new powerdomain. Use this function on the denotation of a predicate symbol or term in the Smyth powerdomain.

The following lemma says that we get the same result in both cases. We use $\mathcal{E}[\![]\!]$ ambiguously for both definitions, the actual use being indicated by the context: both $\mathcal{E}[\![]\!]$ on the left hand sides are elements of some suitable Smyth powerdomain construction and all $\mathcal{E}[\![]\!]$ on the right hand sides are elements of the corresponding new powerdomain construction.

Lemma 30 *Let p be an n-ary predicate. Then*

1. $r(\mathcal{E}[\![t]\!]) = \mathcal{E}[\![t]\!]$
2. $r(\mathcal{E}[\![p]\!]) = \mathcal{E}[\![p]\!]$
3. $r(\mathcal{E}[\![p(\vec{u}]\!]) = \mathcal{E}[\![p(\vec{u})]\!]$

Proof: Simple induction on the order of definition of the denotations using lemma 29. ∎