Investigations Into Abstraction and Concurrency

Radhakrishnan Jagadeesan
Ph.D Thesis

91-1231
August 1991

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

INVESTIGATIONS INTO ABSTRACTION AND CONCURRENCY

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Radhakrishnan Jagadeesan

August 1991

Biographical Sketch

Radhakrishnan Jagadeesan was born as J. Radhakrishnan on 9 November, 1966 in Cuddapah, India. His first ten years of schooling were at G.R.G Higher Secondary School, Coimbatore. He completed his last two years of schooling at St. Mary's Senior Secondary School, Madras. He enrolled in the undergraduate program at the Indian Institute of Technology, Kanpur, and graduated in 1987 with a Bachelor of Technology degree in Computer Science. He became a graduate student in Computer Science at Cornell University in August 1987. Yielding to the pleas of mercy of his friends in Ithaca, he intends to graduate with a Ph.D degree in Computer Science in August 1991.

To Mother, Father, Alka and Shubha.

Acknowledgements

I would like to thank Prakash Panangaden for being a great thesis advisor. His advice, on matters academic and otherwise, have been influential throughout my stay in Cornell. This thesis would not have been started or finished without his ideas, encouragement and support.

It is a pleasure to acknowledge the contribution of Keshav Pingali to this thesis. Indeed, a significant portion of this thesis is a mere explication of his semantic insights.

I wish to thank B. Bloom, D. Kozen and A. Nerode for serving on my thesis committee. I have benefitted by discussions with S. Abramsky, L. Aceto, R. Alur, B. Bloom, T. D. Chandra, R. DiCosmo, C. Gunter, E. Gunter, D. Kozen, G. Lauri, W. Li, G. Lindstrom, A. Jung, J. Prasad, D. Ranjan, J. Reppy, J. Russell, V. Shanbhogue, E. Stark, D. Subramanian and V. Saraswat.

This would be incomplete without expressing my gratitude to the Computer Science Department, my friends in Ithaca and elsewhere, to whom I owe more than can be expressed in words.

Table of Contents

List of Figures

Chapter 1

Introduction

Abstract semantics serves as an interface between a language and the user: the interface serving to hide a significant degree of operational detail and present a simpler view of program execution to the user. What are the properties that an abstract semantics should satisfy? It was an insight of Strachey [41] that the abstract semantics must be compositional; thus, one must be able to "build up" the abstract meaning of a program from the abstract meanings of its components. Note that it is not immediately obvious how one thinks of recursive programs in this manner. This is the role of fixpoint theory developed by Scott [62]. Informally, fixpoint theory formalisms the intuitive inductive arguments that one uses in reasoning about recursive programs. The theory developed by Scott works for determinate programs: programs that yield one output for every input. Thus, the theory is inadequate to handle indeterminate programs: programs that can yield more than one output for a given input. Powerdomains, that can be viewed as the computable analogue of the powerset, were developed to enlarge the scope of the theory of Scott to handle indeterminate computations [52,64]. Domain theory, enriched with powerdomain constructions to handle indeterminacy has been successful in serving as a mathematical formalism powerful enough to specify abstract semantics for transformational programs: programs that take a input, compute in isolation and return an output. Thus, programs are usually denoted by functions in the determinate case and as Input-Output relations in the indeterminate case. The semantics is termed abstract because it hides a significant degree of internal operational detail and presents an extensional view of programs.

The situation is not as clear for interactive programming systems: systems built out of processes which engage in communication with the environment and/or other processes while computing. The key difference from transformational systems is that the output produced by a process can influence its input. Static, determinate dataflow [32] is the prime example of the few models interactive systems that are amenable to extensional treatment.

Static Determinate Dataflow

A dataflow net [32,33] is a directed graph, with autonomous computing agents at the nodes of the graph. Each arc is a unidirectional, unbounded FIFO queue, and message transmission takes place from one agent to another in the direction of the arc. The communication is asynchronous. The semantics [32,33] is based on viewing processes as continuous stream functions. Thus, each process is viewed as a stream function from its input channels to its output channels. The behaviors of networks is built up from the abstract meanings of the processes. Given a network in which every process is associated with a stream function, and every channel is associated with a stream variable, the syntactic description of the network induces a set of equations that describe the relationship between these variables. These equations describe the relationship between input and output streams for the nodes of the network. For example, consider the network in Figure 1.1. We can write down the equations:

$$\begin{aligned} ab &= f_A(i) \\ bc &= f_B(ab) \\ \langle ca, o\rangle &= f_C(bc) \end{aligned}$$

Given some fixed streams on the input of the input streams, we can solve these sets of equations to find out the values of the output streams. Note that in the presence of feedback loops, as for example in the network in the figure, we need to use the semantic techniques of Scott. Thus, the set of equations abstract away completely the internal operational details of processes. However, one needs to relate the operational and abstract views of processes. It can be proved that given fixed streams on the input channels of the network, the least solution, in the prefix ordering on streams, accurately describes the operational behavior of the network. For example, in the

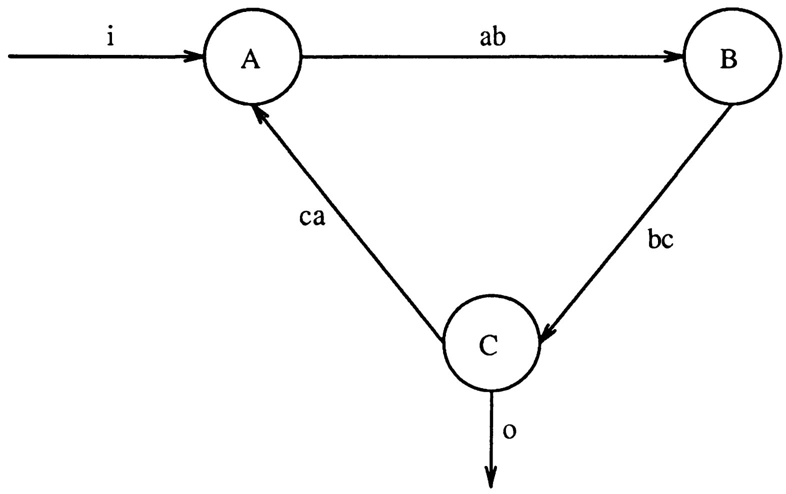


Figure 1.1: A dataflow network

above network, given a fixed stream i, the equations can be solved to deduce a value for o. The claim is that executing the network operationally yields the same stream o. This is Kahn's principle [32,38]. In fact, the correspondence is even tighter. Consider two networks with the same external interface: thus, they have the same number of input channels and the same number of output channels. The natural question to ask if if these networks are intersubstitutable. It can be proved that the two networks are intersubstitutable precisely when their abstract meanings as stream functions are identical. Last but not least, the description of the semantics in the framework of Scott enables the use of fixpoint induction [39] for reasoning about infinite computations.

Unfortunately, the model is not expressive enough. Firstly, the configurations of networks are fixed and cannot change dynamically. For example, the interconnections in the network in Figure 1.1 are fixed. Thus, process B cannot add an output channel to A in runtime. Secondly, the nodes are restricted to be determinate. Indeterminacy is a fundamental property of concurrent systems: arising naturally when one hides internal algorithmic details or timing considerations. Thirdly, notions of higher order abstraction that have arisen in the context of functional programming do not have any obvious analogues in the context of dataflow. This has prompted researchers to investigate more elaborate and expressive models of concurrent computation. However, these elaborations do not share the most attractive feature of the Kahn model: the conceptually simple abstract view of programs. Below, we describe the semantic treatments of some of these elaborations[1].

Indeterminate dataflow

Indeterminate dataflow is the Kahn model enriched with indeterminacy: thus, the nodes are allowed to compute relations instead of functions as in determinate dataflow[2]. Recent results show that Input-Output relations fail to be compositional even in the presence of very mild forms of indeterminacy [55,59,11]. This indicates that there is no hope of attaining the abstract view of programs provided by the Kahn semantics for static determinate dataflow. Also, this result shows that some amount of "extra" information is necessary to describe a compositional semantics of indeterminate dataflow. This extra information takes different forms depending on the extent of indeterminacy allowed. It has been shown that the set of traces of a process describes accurately and completely the behavior of a process [31,34,58] in all contexts. Informally traces can be thought of as encoding interleavings of events on channels on all computation paths. Thus, traces are a significantly less abstract view of programs than for example, the Input-Output relation. There are more abstract models when one "restricts" the indeterminacy that a process can exhibit. Thus, for oraclizable[3] networks , the semantics can be presented as a set of functions [59,2]. Misra describes an equational theory for reasoning about networks of indeterminate processes [46]. All these papers result in considerable simplification in reasoning about processes. However, they do not provide the inductive principles to reason about infinite computations.

Process algebra

Process algebra [43,23] generalizes static dataflow by allowing both indeterminacy and non-directional many to many communication. In contrast to dataflow, process calculi are based on synchronized communication. There is a concurrent composition operation: concurrent composition is a binary operation that composes two agents and allows them to communicate. Recent research has also shown how to add " mobility" to

[1]This review is not intended to be comprehensive or complete, and is intended only to give the reader a feel for the state of research into the semantics of concurrent systems.

[2]In the light of the discussion above, it is worthwhile to note that the configurations of networks is still static in this model.

[3]A network is oraclizable if the indeterminacy can be restricted to a single node: a number spitter without inputs and one output on which it outputs an arbitrary infinite sequence of natural numbers

process algebra [45]. Thus, one can describe processes which can be arbitrarily linked; furthermore, a communication between neighbors can cause change in the structure of linkage of the network. The semantic treatment of process calculi is based on identifying the notion of observation: the behavior seen by an external observer. Two systems are then said to be identical if their observations are identical. There are varying ideas on the exact notion of observations [43,14,9,20,24]. However, for all these notions of observation, there is a rich set of tools to reason about processes. In particular, there is an algebraic theory. In fact, this equational theory is complete for reasoning about finite processes [22,20,9]: processes defined without recursion. There are also modal logics to support proofs of logical assertions about processes [44, 9]. Again, these proof systems are complete for reasoning about finite processes. In spite of the considerable sophistication of these methods, the study remains essentially an operational analysis of processes. Also, the theory does not mesh well with the fixpoint reasoning that is needed to reason about infinite computations. In particular, the class of recursive programs that admit a proof technique are those that possess unique solutions [43]. Note that the induction principles based on domain theory work without assuming unique solutions. This has motivated research into models based on the semantic ideas of Scott. However, there is a significant mismatch between the view of programs provided by standard domain theoretic models of indeterminacy and the view of programs taken by the operational semantics [22]. Consequently, the domain theoretic models are either constructed from the terms of the calculus [22,69] or from computation sequences [24].

Overview of thesis

Abstract semantics simplifies the programmers view of processes and motivates simple proof rules, thus making programs easier to design and verify. This motivates the study of abstract semantics for computational models of concurrency and parallelism. This thesis investigates abstract models for some elaborations of the model of static dataflow. These abstract models will be built in the framework of domain theory. Thus, all the models constructed here will be compositional and support induction principles. The common operational properties of the languages studied here can be expressed informally in terms of the transition relation as follows. This will help to delineate the scope of the investigations in this thesis.

- Enabled input transitions are never disabled. Thus, the languages studied do not have interrupts.
- The languages have blocking reads i.e processes that read a channel for values wait till values appear on the channel;
- Writing is asynchronous, i.e processes do not synchronism on output transitions.
- Indeterminacy, if any, is finite, i.e the execution tree of a process can be expressed as a finitely branching tree. Thus, the thesis does not address fairness issues, as fairness is known to cause unbounded indeterminacy, see for example [15].

In the light of the previous discussion on alternate elaborations of the dataflow model, it should be noted that both indeterminate dataflow and process calculi are expressive enough to have interrupts.

The first section of the thesis describes a declarative semantics for a functional language with logic variables, Id [49,5][4]. This language can be thought of as a formalism for determinate dynamic dataflow enriched with a notion of shared memory. From the parallel programming point of view, the language offers the ability to share and incrementally define data-structures. From the concurrent programming point of view, the language is expressive enough to allow dynamic[runtime] changes in the connections of the underlying dataflow network. The information in the shared memory increases monotonically as computation proceeds. Since the language is intended for parallel execution, reasoning about the operational semantics involves reasoning about complex interleavings of many threads of computation. The denotational semantics views computation as the process of imposing constraints. The semantic description of this chapter extends the equation-solving paradigm that underlies Kahn semantics for dataflow networks [32] to a more expressive setting in which processes manipulate shared memory locations. Message transmission in dataflow networks is monotonic in the sense that a message cannot be recalled once it has been sent. The framework in this chapter extends this model with monotonic shared memory. This is made possible

[4]The semantics of first order Id is joint work with Keshav Pingali and Prakash Panangaden. The semantics of the higher order language is joint work with Keshav Pingali

by modeling constraints as closure operators, an idea due to Keshav Pingali. This extension is powerful enough to allow the communication abilities of processes to change dynamically. The semantics takes the form of equation solving and abstracts away from the interleaving details of the operational semantics. Tight correspondence results are proved between the denotational and operational view of processes. For the first order fragment of the language, it is shown that the denotational semantics is sound and complete for reasoning about the equality of programs. For the full language, the denotational semantics is shown to be sound for reasoning about the equality of programs.

The second section of the thesis[5] constructs a model for a process calculus, the γ calculus [10,8] in which processes are first class values. The model is constructed domain-theoretically and is based on viewing the communication ability of processes as the basic observable. For example, a process that diverges has no communication ability, and a process that accepts one input and then diverges has more communication ability. The main contribution of this section of the thesis is a new powerdomain construction. Roughly speaking, the new construction allows us to model processes as sets of functions. Intuitively, the key difference between our powerdomain and traditional powerdomains [52,64] is that the construction is defined to work on function spaces. Thus, sets of functions that yield the same sets of results on all inputs are equated in the model. In particular, the ordering is coarser than the orderings of the usual powerdomain constructions on function spaces. The model so constructed recovers some algebraic laws satisfied by processes, that models based on usual powerdomain constructions do not satisfy. In particular, when one attempts to use the Plotkin powerdomain to model the γ calculus, one finds that certain operational laws are violated in the model. Concretely, there are terms that are deemed equal by the operational semantics, such that the meanings of these terms are unrelated in a model based on the Plotkin powerdomain, but which are identified in the semantics based on the new powerdomain. In fact, we prove that the model that we construct is "internally fully abstract" [4]. More precisely, we can define an operational preorder on terms that embodies the above notions of observability. It turns out that this preorder meshes

[5]Joint work with Prakash Panangaden

well with the partial order in our model in the following sense. Suppose that we define a preorder on domain elements that formally imitates the definition of the operational preorder. Then, we recover exactly the original partial order in the model. This does not happen with the Plotkin powerdomain.

The third section of the thesis[6] combines the tools of the first two sections to describe an abstract semantics for systems of concurrent (constraint) logic programming. The key difference from pure logic programming is the use of logic variables for synchronization through blocking reads: a process that reads an uninitialized logic variable waits until the variable is instantiated. Concurrent logic programming generalists Kahn's model of static dataflow by admitting indeterminacy through OR-parallelism, and by allowing the communication abilities of processes to change dynamically. The semantics in this chapter 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 treats error in computations as "benign", and thus identifies programs that are distinguishable by existing reasonable notions of observations [17,57,13]. In particular, the semantics is inadequate for studying "progress properties". The results relating the operational and denotational semantics identifies *precisely* the notion of distinguishability that the semantics captures. The aim of the treatment here is to simplify the programmer's view of processes and is intended as a programmer's *first* approximation in thinking about processes. We hope to convince the reader that the view of observability modeled by the semantics is a significant and non-trivial subset of the observations modeled in previous work.

[6]Joint work with Vasant Shanbhogue

Chapter 2

Id: A functional language with logic variables

This part of the thesis[1] presents an abstract denotational semantics for Id [49,5], a functional language, equipped with a notion of shared memory, in the form of logic variables. The operational semantics involves reduction (as in functional languages), synchronization (through blocking reads of logic variables) and constraint-solving (through unification). The subtle interaction between concurrency and logic variables makes the operational semantics far too complex for reasoning about program behavior. This motivates the study of an abstract semantics. The abstract denotational semantics for Id views programs as constraints and is presented in terms of equation solving. This is made possible by modeling constraints as closure operators, an idea due to Keshav Pingali. Of course, one has to prove that the denotational semantics corresponds in a sensible way to the operational semantics. This section develops the technical tools for recovering operational interleavings from semantic information. Based on these tools, a full abstraction result is proved for the first order fragment of the language and an adequacy result is proved for the full language.

The techniques of this section are of interest to two distinct lines of research.

Functional Programming: Id developed as a product of research into the integration of logic variables in functional languages. The merits of this scheme has

[1]The results on the first order fragment are drawn from joint work with Keshav Pingali and Prakash Panangaden [27]. The results for the full language is joint work with Keshav Pingali [28]

been studied extensively, see for example [36,56,61]. Logic variables permit incremental definition of data structures without the excessive copying overhead associated with usual functional languages. For example, logic variables permit elegant coding of constraint-based algorithms such as Milner's polymorphic type deduction algorithm and of symbol-table management algorithms in compilers [5].

Logic Programming: Researchers in concurrent[constraint][logic] programming have investigated the use of these paradigms for the specification and implementation of systems programs [67,60,63]. For the purposes of this section, the key insight is the use of logic variables for synchronization. Informally, this is achieved by making a process reading a variable wait until the variable is instantiated. Indeed, recent work shows that these paradigms subsume determinate dataflow [63,57]. These papers also show that the presence of logic variables allows two features that are lacking in the traditional framework of determinate dataflow. Channels become bidirectional conduits of information as opposed to the unidirectional point-to point entities in the Kahn framework. Furthermore, the ability to pass around variable names allows processes to pass around their communication capabilities.

Organization of chapter

This chapter is organized as follows.

The formal study of the first order fragment is presented first. A few informal examples are used to introduce the language. The operational semantics is presented formally in two styles: first as a structured operational semantics [54] and then in the style of the Chemical Abstract machine, to highlight the inherent concurrency of the operational model. The denotational semantics identifies closure operators [62] as the domain-theoretic models of constraints. A full-abstraction result is then proved that shows that the denotational semantics matches the operational semantics accurately. The key technical tool in proving this relationship is the *Interleaving lemma* (Lemma 10). This lemma recovers the operational notions of interleaving, aliasing and synchronization from semantic information.

The presentation of the development for the full language follows the treatment of the first order fragment. An informal example is used to illustrate the subtlety of the higher order language. This example also serves as evidence that the usual techniques to handle higher order functions are not adequate to model the full language. In particular, higher-order functions can interact with logic variables in complicated ways to give rise to behavior reminiscent of own variables in Algol-60 [48]. The operational semantics is presented formally, followed by the abstract semantics. The semantics for the full language is an extension of the semantics for the first order fragment. When restricted to the first order fragment, the semantics cuts down to the original first order semantics. A standard extension of the tools developed in the study of the first order language is then used to show that the denotational semantics is adequate with respect to the operational semantics, i.e the abstract semantics is correct for reasoning about equality in the operational semantics. These results show that a higher-order functional language with logic variables can be viewed as a language of incremental definition of functions.

2.1 Cid: the first order language

This section is the study of the first order fragment of Id, henceforth called Cid. A few informal examples are used to introduce the language. Next, the operational semantics is presented formally, followed by the denotational semantics. Finally the proof of full-abstraction of the denotational semantics is presented.

2.1.1 Informal Introduction to language

The language is introduced through a series of three programming examples. The first example in this section illustrates the definition and use of logical arrays. The next two examples illustrate the inherent multi-threadedness of the language.

Array constructs

There are three constructs for handling logical arrays and respectively serve for allocating, storing into and reading from arrays.

1) An array is allocated by an expression of the form $arrays(e)$, where e is an expression that must evaluate to a positive integer. The array can be named via a definition;

for example, the definition $A = array(5)$ allocates an array of length 5 and names it `A`. When an array is allocated, its elements are undefined and each element of the array is an uninstantiated logic variable.

2) An element of an array A can be given a value by a statement of the form: $A[i] = v$. On execution, the value v is unified with the value contained in $A[i]$ and the resulting value is stored into $A[i]$. If unification fails, the entire program is considered to be in error.

3) An element of an array is selected by using the expression $A[i]$.

The following simple example illustrates these ideas. Note that the semi-colon below indicates parallel composition, and not sequential composition. Thus, the order of statements is irrelevant and the semicolon should be thought of as a separator of elements of a multiset. The function $add - boundary$ below takes as input an array of length n and creates an array of size $n + 2$ whose end elements are 0's.

```
def add-boundary(B,n) =
                        {A = array(n+2);
                         for i from 2 to n+1 do
                           A[i] = B[i-1]
                         A[1] =0;
                         A[n+2]=0;
                         in A}
```

Consider the execution of the call `add-boundary([1,2,3], 3)`. Substituting actuals for formals, we get:

```
{ A = array(5);
  for i from 2 to 4 do
    A[i] = [1,2,3] [i-1]
  od;
  A[1] =0 ;
  A[5]=0;
  in A}
```

Execution of $array(5)$ returns an ordered list of 5 undefined logic variables. Applying this rule and replacing the for loop by the set of statements corresponding to each loop index, we get:

```
{ A = [L1,L2,L3,L4,L5];
  A[2] = [1,2,3] [1];
  A[3] = [1,2,3] [2];
  A[4] = [1,2,3] [3];
  A[5]=0;
  A[1] =0 ;
  in A}
```

$[L1 \ldots Ln][i]$ reduces to Li. Using this rule on either side of the equations above, we get:

```
{  A = [L1,L2,L3,L4,L5];
   L2 = 1;
   L3 = 2;
   L4 = 3;
   L5=0;
   L1 =0 ;
   in A}
```

Thus we get the result $[0, 1, 2, 3, 0]$. Note that the array A has been created by the cooperation of many definitions. Abstractly, this process can be viewed in terms of constraint intersection. Thus, $A = array(5)$ constrains A to be an array of size 5. Similarly, $A[5] = 0$ imposes a constraint on the 5th element of A and so on. The resulting value of A is obtained by the conjunction of these constraints.

Multi-threadedness

The above program could have been executed sequentially. In general, an Id Nouveau program cannot be executed sequentially. This is indicated by the following example, that is written in pseudo-code. The example also illustrates the use of blocking reads of logic variables to achieve synchronization.

```
{A = array(n); B = array(n);
```

```
     producer(A,B,1);
     consumer(A,B,1)
    }
```

Arrays A and B establish channels between the producer processe and the consumer process. A is used by the producer to communicate values to the consumer. B is used by the consumer to send back acknowledgments to the producer, and to indicate if the consumer expects more values. The *producer* outputs a value on the present location $A[i]$ and waits for a signal from the consumer on $B[i]$ to indicate continuation. The waiting occurs because the conditional blocks until its guard reduces to a base value. $B[i] = 0$ indicates continuation and other values of $B[i]$ indicate termination.

```
def producer(A,B,i) ={
        v =  .. compute v ....
        A[i]  = v ;
        in
        cond( B[i] = 0, producer(A,B,i+1), 1)}

def consumer(A,B,i) = {
       more_vals_needed = .......
       in
       cond(more_vals_needed,B[i]=0 in consumer(A,B,i+1),B[i]=1 in 1)
}
```

The main point of this example is that it cannot be executed sequentially. The *producer* and *consumer* processes are two distinct threads of execution that must be executed in parallel. Fortunately, the viewpoint of constraints provides a way to mask the operational complexity - we can think of the *producer* and *consumer* as constraining the arrays A and B, and the end result is produced by the conjunction of these constraints with the constraints $A = array(n); B = array(n)$. These ideas are formalized in the denotational semantics.

The following example encodes the "parallel or" function. The "parallel or" function is a binary function from boolean to boolean that returns true if either of its arguments is true, and returns false only if both its arguments are false. Plotkin's work on full abstraction in LCF [53] shows that this function is not definable in LCF. The proof relies on the "sequentiality" of the operational semantics of the lambda calculus. In this light, the ability to code the function in the first order fragment of the language can be viewed as semantic evidence for the inherent multithreadedness of the language. The idea is that the result of the function *res*, is instantiated by three simultaneously executing threads: each of which can be thought of as a line in the truth table of "parallel or". Note that the function returns an uninstantiated variable if both arguments are undefined or if one argument is false and the other is uninstantiated.

```
def parallel-or(x,y) = {
          res = array(1);
          res = if x = true then true else array(1);
          res = if y = true then true else array(1);
          res = if (x = false and y =false) then false else array(1);
          in
          res
           }
```

2.1.2 Syntax

The formal syntax of Cid is presented in figure 2.1. To simplify notation, it is assumed that all procedures return a result. It is also assumed that the result of execution of the main program is the binding of a special variable x. The body of a procedure is a single scope and the formal parameters of the procedure are in the same scope. For notational convenience, it is required that a procedure have exactly one formal parameter. In the diagram, *op* is a generic symbol for binary operations on arrays. These include array selection $e_1[e_2]$, arithmetic operations like multiplication, addition and so on.

Next, the operational semantics of Cid is presented in two styles: first in a Plotkin-style [54] structured operational semantics, and secondly in the setting of the Chemical Abstract Machine [8].

program	::=	
		def F1(id) = def-list in exp
		def F2(id) = def-list in exp
		....
		def Fn(id) = def-list in exp
		x = exp
def-list	::=	def | def;def-list
def	::=	id = exp
expression	::=	constant | id | exp1 op exp2 |
		cond(exp1, exp2, exp3) |
		array(exp) | Fi(exp1)

Figure 2.1: Syntax of Cid

2.1.3 Structured Operational Semantics of Cid

In this section, we give an operational semantics for Id using Plotkin-style [54] state transition rules. Rather than rewrite expressions directly, it is convenient to work with *configurations.* A configuration is a tuple $< D, e, \rho, FL >$ — intuitively, D contains definitions whose right-hand sides have not yet been completely reduced to an identifier, constant, or array. The expression e in the configuration is the expression whose value is to be produced as the result of the program. Configurations are rewritten by reduction and by constraint solving. For example, an occurrence of $2 + 3$ in D or in e can be replaced by 5 in a reduction step. Once the right-hand side of a definition in D has been reduced completely, the definition can participate in constraint solving. Configurations have a component named ρ which keeps track of such definitions. The component ρ, called the environment, keeps track of bindings between identifiers and *base values* (identifiers, constants and arrays) and has a complex structure to permit unification — it consists of a (possibly empty) set of *alias-sets* where an alias-set is an equivalence class of base values. For example, $\{x, y, z\}, \{x, y, 4\}$ and $\{x, y, [L1, L2]\}$ are alias-sets. If $b1$ and $b2$ are two base values in the same alias-set, then occurrences of $b1$ in D and e may be replaced by $b2$ without changing the meaning of the program. If unification fails, the configuration is rewritten to 'Error' and computation aborts. Otherwise, the resulting environment replaces the old one in the configuration, and rewriting continues.

We define some syntactic categories required for the operational semantics.

$x, L \in Id$ = identifiers	$c \in Constant$ = set of constants
$Ar \in Array ::= [x_1 \dots x_n]$	$D \in Defs ::= \emptyset \vert \mathrm{def}_1, \dots \mathrm{def}_n$
$e \in expression$	$\rho \in Environment ::= \emptyset \vert \{A_1, ..., A_n\}$
$B \in Base-value ::= x \vert c \vert Ar$	$A \in$ Alias-set $::= \{B_1, \dots B_n\}$
$FL \in Free-list = \mathcal{P}(\mathrm{Id})$	$C, conf \in$ Config. $::= < D, e, \rho, FL > \vert Error$

We assume that the set of identifiers Id is a countable infinite set. The notation $[x_1, ..., x_n]$ for arrays represents a sequence of n identifiers, where n is greater than or equal to 1. The *length* of an array is the number of elements in this sequence. Also, the definitions D are those defined by language syntax.

Unification

The unification algorithm that we use is similar to the one in Qute [61]. This is an algorithm for the unification problem in the domain of regular infinite trees. Hence, no occurs check is performed and infinite data structures are considered to be legitimate objects of computation.

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

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

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

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

Intuitively, these two transformations on environments leave the meaning of an environment unchanged. 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 $\rightsquigarrow$. It is known [61] that $\overset{*}{\rightsquigarrow}$ is Church-Rosser and Noetherian(terminating). Let $\mathcal{U}(\rho, A)$ denote the unique, irreducible environment ρ_1 such that $(\rho \cup \{A\}) \overset{*}{\rightsquigarrow} \rho_1$.

Rewrite Rules

The rewrite rules for configurations are specified in terms of a binary relation $\rightarrow$ on the set of configurations. In any program P, let exp_P be the expression to be evaluated. The initial configuration for program P is $< \emptyset, exp_P, \emptyset, Id >$.

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

Definition 3 *If $< D, e, \rho, FL >$ is a configuration and x is an identifier not a member of FL, let A be the (unique) alias-set that contains x. The function $\rho(x)$ is defined by cases on A:*

1. *All the elements of A are identifiers. In that case, $\rho(x)$ is undefined.*
2. *At least one element of A is a constant c. Since A is consistent, the elements of A are either identifiers or the constant c. We define $\rho(x)$ to be c.*
3. *At least one element of A is an array. Since A is consistent, the elements of A are either identifiers or arrays of the same length. $\rho(x)$ could be defined to be any one of these arrays. To be precise, place a lexicographical ordering on identifiers and let $\rho(x)$ be the array whose first element is the least in this ordering.*

For notational convenience, we shall assume that $\rho(c) = c$, for constants c. Thus, we will say, $\rho([L_1 \ldots L_n]) = [L_1 \ldots L_n], \rho(3) = 3$ and so on.

The Plotkin-style operational semantics [54] for Cid is given in Figures 2.2 and 2.3 and 2.4. Figure 2.2 describes the rewrites of non-array expressions. During the rewrite process, free occurrences of an identifier x will be replaced by $\rho(x)$ if $\rho(x)$ is defined. Arbitrary contexts are denoted by $C[]$ in this rule. This rule, together with the first rule for definitions, ensures that a free occurrence of an identifier in a configuration can be replaced by the value associated with the identifier in the environment or functional environment. Function application cannot be done by simple substitution as in usual functional languages. Consider the function

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

When `F` is passed an array, it stores 1 and 2 into the first and second components of the array. Consider the expression `F(array(2))`. If `array(2)` is simply substituted for x in the body of the body of the function, the resulting expression is quite different from what one gets by first reducing `array(2)` to a base value and then performing the substitution. A function application $F(e)$ is rewritten by replacing it with $body_F$, and adding the definitions in defs_F to the definitions in D, after renaming the formal parameters and the local variables to avoid name clashes. Since the actual parameter `e` need not be a base value, a definition `x = e` is added to the definitions in the configuration.

Figure 2.3 gives the rewrite rules for array expressions. The *op* used is a generic binary operator. Thus, the rule is used for array selection $e_1[e_2]$, arithmetic operations like addition etc. The difference between these operations arises in the cases when e_1 *op* e_2 reduces to a base value. Thus, for array expressions $e_1[e_2]$ is a base value if e_1 is an array and e_2 is an integer; $e_1 + e_2$ is defined when both the expressions are integers and so on.

Properties of Rewrite Rules

It is straight-forward to prove a Church-Rosser theorem about the rewrite rules in Figures 2.2 and 2.3 and 2.4. The proof reduces to showing that Cid has a 'one-step' Church-Rosser property from which the desired theorem follows by 'pasting together diamonds' as in proofs of the Church-Rosser theorem for lambda-calculus [6]. More precisely, we have the following development.

Definition 4 *$\langle D_1, e_1, \rho_1, FL_1\rangle$, $\langle D_2, e_2, \rho_2, FL_2\rangle$ are* alpha-*equivalent if $\exists x_1 \ldots x_n \in FL_1, y_1 \ldots y_n \in FL_2$, such that FL_1 - $\{x_1 \ldots x_n\}$ $=$ FL_2 - $\{y_1 \ldots y_n\}$, and replacing $x_1 \ldots x_n$ by $y_1 \ldots y_n$ in D_1, e_1, ρ_1 gives D_2, e_2, ρ_2 respectively.*

We assume the existence of a $\xrightarrow{\alpha}$ rule [6]. The following lemma says essentially that Cid has a one-step Church-Rosser property. It can be viewed as saying that two enabled reductions do not interfere with each other.

Lemma 1 *Let $\langle D_0, e_0, \rho_0, FL_0\rangle \rightarrow conf_1$, $\langle D_0, e_0, \rho_0, FL_0\rangle \rightarrow conf_2$. Then, one of the following holds:*

1. *If $conf_1 =$ error , $conf_2 \rightarrow$ error*
2. *If $conf_2 =$ error , $conf_1 \rightarrow$ error*

Var:	1. $< D, C[x], \rho, FL > \ \rightarrow\ < D, C[\rho(x)/x], \rho, FL >$ (if $\rho(x)$ defined)
Cond:	1. $< D, cond(e_1, e_2, e_3), \rho, FL > \ \rightarrow\ < D^*, cond(x_1, e_2, e_3), \rho^*, FL^* >$
	$x_1 \in FL$, $\rho^* = \rho \cup \{\{x_1\}\}$, $FL^* = FL - \{x_1\}$, $D^* = D \cup \{x_1 = e_1\}$
	2. $< D, cond(true, e_2, e_3), \rho, FL > \ \rightarrow\ < D, e_2, \rho, FL >$
	3. $< D, cond(false, e_2, e_3), \rho, FL > \ \rightarrow\ < D, e_3, \rho, FL >$
Appl:	1. $< D, F(e), \rho, FL > \ \rightarrow\ < D, x = e \textbf{ in } e_1, \rho^*, FL^* >$
	where $e_1 = body_F[x/arg_F][y/local_F]$
	$x, y \epsilon FL$, $\rho^* = \rho \cup \{\{x\}, \{y\}\}$, $FL^* = FL$ - $\{x, y\}$
	2. $< D, x = e \textbf{ in } e_1, \rho, FL > \ \rightarrow\ < D \cup \{x = e\}, e_1, \rho, FL >$

Figure 2.2: Structured Operational Semantics of Cid: Expressions

3. *Let* $conf_1 = \langle D_1, e_1, \rho_1, FL_1 \rangle$, $conf_2 = \langle D_2, e_2, \rho_2, FL_2 \rangle$, *and* $(FL_0 - FL_1) \bigcap (FL_0 - FL_2) = \emptyset$. *Then one of the following holds:*
   (a) $conf_1 \xrightarrow{\alpha} conf_2$
   (b) $(\exists conf_3)\ [conf_1 \rightarrow conf_3 \wedge conf_2 \rightarrow conf_3]$

Proof: The proof follows immediately from a case-by-case analysis of the rules in Figures 2.2 and 2.3 and 2.4. ∎

The interpreter rewrites configurations, selecting any enabled transition at each step. To guarantee progress, there must be some notion of fair-scheduling in the sense that no enabled transition is postponed indefinitely.

2.1.4 CHAM Operational semantics of Cid

In this section, the operational semantics for Cid is described in the formalism of the Chemical Abstract Machine [8]. The Chemical Abstract Machine is a formalism for the description of asynchronous concurrent systems. The idea is to model the notion of processes "moving around freely" and communicating when they come into contact. Thus, the system is described by a chemical solution, where the processes are like molecules, that interact on contact. Transformation of solutions is concurrent; because any set of reactions that involve disjoint sets of molecules can be performed at the same time. A program to compute the prime numbers less than a given natural

1. $< D, array(e), \rho, FL > \;\rightarrow\; < D \cup \{x = e\}, array(x), \rho \cup \{\{x_1\}\}, >$

where $x \in FL$, $\rho^* = \rho \cup \{\{x\}\}$, $FL^* = FL - \{x\}$

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

where L1,...,Ln ϵ FL, $\rho^* = \rho \cup \{\{\text{L1}\},...\{\text{Ln}\}\}$, $FL^* = FL - \{L1, ...Ln\}$

3. $< D, e_1 \; op \; e_2, \rho, FL > \;\rightarrow\; < D \cup \{x_1 = e_1, x_2 = e_2\}, x_1 \; op \; x_2, \rho^*, FL^* >$

where $\{x_1, x_2\} \subseteq FL$, $FL^* = FL - \{x_1, x_2\}$, $\rho^* = \rho \cup \{\{x_1\}, \{x_2\}\}$

4. $< D, b_1 \; op \; b_2, \rho, FL > \;\rightarrow\; < D, r, \rho, FL >$

if b_1, b_2 are defined and $r = b_1 \; op \; b_2$ is defined.

Figure 2.3: Structured Operational Semantics of Cid: Array Expressions

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

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

   if v is base value or a variable and $\mathcal{U}(\rho, \{x, v\})$ is consistent

   $< D \cup \{x = v\}, e, \rho, FL > \rightarrow Error$

   if v is base value or a variable and $\mathcal{U}(\rho, \{x, v\})$ is inconsistent

Figure 2.4: Structured Operational Semantics of Cid: Definitions

number n, drawn from the paper of Boudol and Berry [8] will serve to illustrate the concepts. Assume that the solution is initially made up of all integers between 2 and n. There is only one rule: each integer destroys its multiples. Then the solution will end up containing the required prime numbers. Note that one needs a notion of fairness to ensure that no enabled reduction is postponed indefinitely. Thus it is assumed that there is some mechanism to ensure that molecules come into contact.

The motivation for using this presentation is twofold: Firstly, to bring out the inherent parallelism of Cid, and secondly to highlight the local nature of the reduction rules. This presentation also masks the difference between resolved and unresolved constraints made in the structured operational semantics for Cid; this can alternately be stated by saying that the store is decomposed into its constituent primitive constraints. Thus, there will be molecules corresponding to the bindings of variables in the environment and molecules corresponding to the unresolved constraints. Reactions will allow us to mimic the process creation and synchronization features in the language.

This subsection follows the notation of previous work [3]. For the sake of simplicity, we assume that a program is a set of definitions. Thus, we assume that the final result of a program is deposited in a special variable, say res. The relationship between the Structured operational semantics of the previous section and the approach in this section is roughly as follows. A configuration $\langle D, , \rho, FL\rangle$[2] is transformed into a set of equations induced naturally by $D \cup \rho$.

Define the syntactic categories $Term, Equations$ as follows:

$$\begin{array}{rcl} e \in Term & ::= & Var \mid op(e_1, e_2) \mid Array(t) \mid cond(e_1, e_2, e_3) \\ & & \mid F(e) \mid c \mid E; e \\ \Theta, \Xi \in Equations & ::= & \Theta_1, \Theta_2 \mid Var = e \end{array}$$

The rules are presented in Figures 2.5 and 2.6 and 2.7. The rules are of two major kinds.

Structural Rules: These are meta rules that specify the machine and define the relation $\rightleftharpoons$. $\overset{*}{\rightleftharpoons}$ denotes the transitive closure of $\rightleftharpoons$. The first two structural rules capture the fact that the list of equations is a multiset of multisets of size

[2]There is no expression component of the configuration, since we are assuming that a program is set of definitions

1. $e_1 = e_2 \rightleftharpoons e_2 = e_1$
2. $e_1 = e_2, e'_1 = e'_2 \rightleftharpoons e'_1 = e'_2, e_1 = e_2$
3. $\dfrac{\Theta \rightleftharpoons \Theta'}{\Theta, \Xi \rightleftharpoons \Theta', \Xi}$
4. $\dfrac{\Theta \overset{*}{\rightleftharpoons} \Theta', \Theta' \rightarrow \Xi', \Xi' \overset{*}{\rightleftharpoons} \Xi'}{\Theta \rightarrow \Xi}$

Figure 2.5: CHAM semantics of Cid: Structural Rules

two. This brings out the fact that there is no inherent order among the various equations, and among the two constituents of each equation. The penultimate structural rule captures the fact that enabled reductions in a molecule can be performed in any solution. The last rule is the mixing rule.

Reaction Rules: These correspond to actual computation. There are three kinds of reaction rules.

1. The **Process creation** rules spawn off new computations. We assume that there is a suitable scheme to ensure that the variables created in the right hand sides of the process creations rules are new. For example, this can be ensured by associating a unique tag with each equation(process), thus maintaining the locality of the rules.
2. **Synchronization rules** involve interaction between different agents, and indicate the points at which interaction is essential for computation to proceed. Note that the interactions are local to the processes involved.
3. The **Unification rules** perform the updates necessary to keep track of the store. Note that the rules of unification create unnecessary (but harmless) copies of bindings of variables. The assumption of fairness ensures that the semantics is correct. The last rule of this set propagates *error* upward and indicates that detection of error anywhere aborts the entire computation.

2.1.5 Denotational Semantics

The discussion in Section 2.1.1 motivated viewing Cid programs in terms of constraints. Thus, a definition of the form $x = array(3)$ was thought of as a piece of partial infor-

1. $e' = (e_1 \; op \; e_2) \rightarrow e' = (x_1 \; op \; x_2), x_1 = e_1, x_2 = e_2$
2. $e' = Array(e) \rightarrow e' = Array(x), x = e$
3. $e' = F(e) \rightarrow e' = body_F[x/arg_F][y/local_F], x = e$
4. $e' = cond(e_1, e_2, e_3) \rightarrow e' = cond(x, e_2, e_3), x = e_1$
5. $e = (\Theta; e') \rightarrow \Theta, e = e'$
6. $y = e, x = v \rightarrow y = e[v/x], x = v$, where v is a base value.
7. $e' = cond(true, e_2, e_3) \rightarrow e' = e_2$
8. $e' = cond(false, e_2, e_3) \rightarrow e' = e_3$

Figure 2.6: CHAM semantics of Cid: Process Spawn and Synchronization Rules

1. $x = y, y = v \rightarrow x = y, x = v, y = t$, if v is a base value.
2. $[L_1 \dots L_n] = [L'_1 \dots L'_n] \rightarrow L_1 = L'_1, \dots L_n = L'_n$.
3. $b_1 = b_2 \rightarrow error$, if b_1, b_2 inconsistent.
4. $\dfrac{\Theta \rightarrow error}{\Theta, \Xi \rightarrow error}$

Figure 2.7: CHAM semantics of Cid: Unification Rules

mation about x satisfied by any array of length 3. Similarly, the definition $x[1] = 2$, induces a constraint that is satisfied by any array with first element is 2. This section identifies the domain-theoretic model of constraints, and presents a denotational semantics for Cid.

The rest of this section is organized as follows. First, we describe the construction of the domain of values, to provide a formal setting for the discussions in the rest of the section. Next, we identify intuitive properties that hold of constraints. This motivates modeling constraints as closure operators [62]. It is then shown that closure operators satisfy these intuitive criteria. Finally, the semantics of the language is presented.

The Semantic Domain

To define the domain of arrays we use a standard construction for defining a domain of (possibly infinite) terms in logic programming, as described in Lloyd's book [37]. We need some notation to describe the construction. Let ω be the set of natural numbers. We use ω^* for the set of finite sequences of natural numbers. A sequence is written $[i_1, \ldots, i_n]$. If s and t are sequences then $[s,t]$ denotes their concatenation, if s is a sequence and n is a natural number then $[s,n]$ is the sequence s with n added to the end. The size of a set X is written $|X|$ and the size of a sequence s is written $|s|$.

Definition 5 *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 nested. The sequences are the tree addresses of the nodes of the tree. We define $br(s,t)$ to be the number of successors of the node s in the tree t; if the tree is clear from context we will write $br(s)$. If this number is 0, s is a leaf.

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

Definition 6 *An element of W is a function $f : t \rightarrow A$ where t is a non-empty tree. The function f satisfies $\forall s \in t.br(s) = 0 \Rightarrow f(s) \in (Atom \cup \Omega) \wedge br(s) = n \neq 0 \Rightarrow f(s) = array_n$. The ordering between elements of W is defined as follows: $f \sqsubseteq g$ if and only if*

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

The ordering between elements of W allows one to replace occurrences of Ω with other elements to obtain a larger element. This domain describes infinitely deeply nested arrays but all arrays must have finite "width". Note that if two arrays have different widths they are incomparable. Thus the domain decomposes into subdomains corresponding to different array sizes. It is straightforward to check that the V is an algebraic complete lattice. The least element $\perp$ is the completely undefined value (Ω in the above discussion) and the maximum element is $\top$, that models overdefined (inconsistent) values.

Constraints are Closure Operators

This section identifies the domain theoretic models of constraints as closure operators. This idea is due to K. Pingali. As an aid to understanding the domain theoretic model of constraints, consider the intuitive properties that we expect constraints to satisfy .

1. Imposition of a constraint on a variable x increases information about x.
2. The imposition of constraints is an idempotent operation. For example, the effect of imposing the constraint $x[1] = 2$ is equivalent to imposing the constraint once.
3. Constraints are determined uniquely by the set of values that satisfy the constraint.
4. A state satisfies two constraints if and only if it satisfies each constraint individually. Thus, the set of states that satisfy two constraints is the set intersection of the states that satisfy each individual constraint.

Note that it is a corollary of the last item that the operation of imposition of constraints is commutative and associative. Thus, parallel imposition of constraints satisfies the algebraic laws governing parallel composition that investigated by researchers in the theory of concurrency [8,43].

Let D be a complete algebraic lattice. Following Scott's thesis, we expect constraints to be modeled by continuous operators on D. The first two conditions above motivate considering operators f on D that satisfy:

Extensivity: $x \sqsubseteq f(x)$

Idempotence: $f(f(x)) = x$.

Operators satisfying these conditions are called closure operators [62], and have been studied in the context of models of the lambda calculus. Most of the material on the mathematical properties of closure operators presented here is well known [62].

We first show that closure operators satisfy the criteria outlined above, thus strengthening the belief that closure operators are the right model for constraints. Of course, a perfect match with the operational semantics is to be proved, to completely justify the model. This is done later.

The following definition identifies the fixpoints of a closure operator f. From a constraint point of view, the fixpoints can be thought of as values that satisfy the constraint modeled by f.

Definition 7 *Let f be a closure operator on D. Then, x is a* fixpoint *of f if $f(x) = x$. The set of fixpoints of f is denoted $Fix(f)$.*

The following lemma shows that closure operators provide an extensional view of constraints: they are determined completely by their fixpoint sets. Recall that this was condition 3 of the intuitive properties of constraints outlined at the start of this section.

Lemma 2 *(Closure operators are determined by their fixpoint sets)*
Let $S \subseteq D$. Then, S is the fixpoint set of a closure operator if and only if the following conditions are satisfied:

1. $(\forall x)$ $[x \uparrow \cap S]$ *has a least element in S.*

2. S is closed under the least upper bounds of directed sets

Proof: Let f be a closure operator on D. Consider $Fix(f)$. Then,

1. $(\forall x)$ $[x \uparrow \cap S]$ has a least element $f(x) \in Fix(f)$.

2. Let $\{x_i|i\}$ be a chain such that $x_i \in Fix(f)$. Then, $f(\bigsqcup x_i) = \bigsqcup f(x_i) = \bigsqcup x_i$.

Let $S \subseteq D$ satisfy the conditions. Define f_S by $f_S(x) = \sqcap[x \uparrow \cap S]$. It is immediate that f_S is idempotent, extensive and monotone. Let $\{x_i|i\}$ be a chain and $y_i = f_S(x_i)$. From condition 2, $y = \bigsqcup y_i \in S$. From monotonicity of f_S, $y = \bigsqcup f_S(x_i) \sqsubseteq f_S(\bigsqcup x_i)$. But, $[\bigsqcup x_i \sqsubseteq y \wedge \ y \in S] \Rightarrow f_S(\bigsqcup x_i) \sqsubseteq y$. ∎

Next, we define a notion of parallel composition. Recall from the operational semantics that the reductions of a set of unresolved constraints(the D coordinate of the configurations) was a fair interleaving of the reductions of each unresolved constraints (elements of D). The following definition can be operationally interpreted as constructing a specific interleaving of the executions of f, g.

Definition 8 *(Parallel composition of closure operators)*
Let f, g be closure operators on D. Then, the parallel composition of f and g is the closure operator $h = \bigsqcup_i (f \circ g)^i$.

The following lemma clarifies the meaning of the above definition. The first item below states that the actual order of interleaving is unimportant. The second and third items relate the parallel composition to the constraint view and equation solving. In particular, the third item shows that in the constraint view, parallel composition reduces to intersection of the fixpoint sets. We use the notation **lcs** to stand for the *least common solution* of a set of equations.

Lemma 3 *h as defined in definition 8 satisfies:*

1. $h = \bigsqcup_i (f \circ g)^i = \bigsqcup_i (g \circ f)^i$.

2. Forall x, $h(x)$ can be defined as follows:

$$h\ x \ = \ \underset{\text{in } y}{\mathbf{lcs}} \begin{cases} x \sqsubseteq y \\ y = f\ x \\ y = g\ x \end{cases}$$

3. *$Fix(h) = Fix(f) \cap Fix(g)$*

4. *The parallel composition operation is idempotent, commutative and associative.*

Proof: (Sketch)

1. $(f \circ g)^i \sqsubseteq g \circ (f \circ g)^i \circ f = (g \circ f)^{i+1}$.

2. Let $x \sqsubseteq z$ and $h(z) = z$. Then, a simple inductive argument shows that $(f \circ g)^i(x) \sqsubseteq z$, for all i.

The proofs of 3, 4 are straightforward and is omitted. ∎

Finally, the space of closure operators on a complete algebraic lattice, ordered extensionally forms a complete algebraic lattice. The proof of the following lemma is straightforward and is omitted.

Lemma 4 *Let D be a complete algebraic lattice. Let $D \rightarrow_c D$ be the space of closure operators on D. Then, the following hold:*

1. *$D \rightarrow_c D$ is a complete algebraic lattice with binary sup of f, g being the parallel composition of f, g and the limit of chains coinciding with the usual pointwise limits in function spaces.*

2. *$f \sqsubseteq g \Leftrightarrow Fix(g) \subseteq Fix(f)$*

In the sequel, we will be interested in algebraic lattices in which $\top$ is a finite element. The operational meaning of this assumption is the finite detectability of inconsistent constraints. Note that the domain V satisfies this property. We define below the $\top$-strict product domain of algebraic lattices with finite $\top$, D_1 and D_2. This is denoted $D_1 \times_\top D_2$. The definition resembles the usual product structure except that, the top elements are "coalesced", so that the pairing operator is strict with respect to $\top$. We denote the infinite $\top$-strict product of D_i by $\Pi_i^\top D_i$. An example of such an infinite product is the space of semantic environments $ENV = \Pi_x^\top V_x$, where x ranges over variable names. The motivation for the definition of a $\top$-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 9 *Let D_1, D_2 be algebraic lattices with finite- $\top$. $D_1 \times_\top D_2$ is the partial order defined as follows:*

$$\mathcal{C}[\![x = e]\!]\ env \;=\; \underset{env'}{\mathrm{lcs}} \begin{cases} env \sqsubseteq env' \\ \langle env', r\rangle = \mathcal{E}[\![e]\!]\langle env', r\rangle \\ env'[x] = r \end{cases}$$

$$\mathcal{C}[\![def_1\ ;\ def_2]\!]\ env \;=\; \underset{\text{in } env'}{\mathrm{lcs}} \begin{cases} env \sqsubseteq env' \\ env' = \mathcal{C}[\![def_1]\!]\ env' \\ env' = \mathcal{C}[\![def_2]\!]\ env' \end{cases}$$

Figure 2.8: Denotational Semantics of Cid: Definitions

$V \times_{\top} V$. We assume that the symbol F is bound to $\mathcal{F}[\![F]\!]$ in all the environments used in the definition of $\mathcal{E}$.

2.1.6 Relating the Semantic Definitions

In this section we prove that the denotational semantics and the operational semantics coincide. The main result of this section is the full abstraction result: two programs are operationally identical if and only if they have the same denotation. This section is organized as follows: first, we present an outline of the proof; next, we present the proof in full detail.

Proof Outline

This section presents the informal sketch of the proof that the denotational semantics and the operational semantics coincide.

Reduction Preserves Meaning

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

$$\mathcal{C}[\![x = e]\!]\ env \quad = \quad \text{lcs}\begin{cases} env \sqsubseteq env' \\ \langle env', r\rangle = \mathcal{E}[\![e]\!]\langle env', r\rangle \\ env'[x] = r \end{cases}$$
$$env'$$

$$\mathcal{C}[\![def_1\ ;\ def_2]\!]\ env \quad = \quad \text{lcs}\begin{cases} env \sqsubseteq env' \\ env' = \mathcal{C}[\![def_1]\!]\ env' \\ env' = \mathcal{C}[\![def_2]\!]\ env' \end{cases}$$
$$\text{in } env'$$

Figure 2.8: Denotational Semantics of Cid: Definitions

$V \times_{\top} V$. We assume that the symbol F is bound to $\mathcal{F}[\![F]\!]$ in all the environments used in the definition of $\mathcal{E}$.

2.1.6 Relating the Semantic Definitions

In this section we prove that the denotational semantics and the operational semantics coincide. The main result of this section is the full abstraction result: two programs are operationally identical if and only if they have the same denotation. This section is organized as follows: first, we present an outline of the proof; next, we present the proof in full detail.

Proof Outline

This section presents the informal sketch of the proof that the denotational semantics and the operational semantics coincide.

Reduction Preserves Meaning

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

$$\mathcal{E}[\![array(e)]\!]\langle env, a\rangle = \textbf{lcs}\begin{cases} env \sqsubseteq env' \\ \langle env', n\rangle = \mathcal{E}[\![e]\!]\langle env', n\rangle \\ r = \mathcal{A}rray(n) \sqcup a \end{cases}$$
$$\textbf{in } \langle env', r\rangle$$

$$\mathcal{E}[\![[L1, L2, \ldots, Ln]]\!]\langle env, a\rangle = \textbf{lcs}\begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r \\ env'[L1] = r[1] \\ \ldots \\ env'[Ln] = r[n] \end{cases}$$

$$\mathcal{E}[\![e_1 \; op \; e_2]\!]\langle env, a\rangle = \textbf{lcs}\begin{cases} env \sqsubseteq env' \\ \langle env', v_1\rangle = \mathcal{E}[\![e_1]\!]\langle env', v_1\rangle \\ \langle env', v_2\rangle = \mathcal{E}[\![e_2]\!]\langle env', v_2\rangle \\ r = (v_1 \; op \; v_2) \sqcup a \end{cases}$$
$$\textbf{in } \langle env', r\rangle$$

Figure 2.9: Denotational Semantics of Cid: Array Expressions

$$
\begin{aligned}
\mathcal{E}[\![const]\!]\langle env, a\rangle &= \langle env, \mathcal{K}(const) \sqcup a\rangle \\
\mathcal{E}[\![x]\!]\langle env, a\rangle &= \langle env[x \mapsto (env(x) \sqcup a)], env(x) \sqcup a\rangle \\
\mathcal{E}[\![cond(e_1, e_2, e_3)]\!]\langle env, a\rangle &= \textbf{lcs} \begin{cases} a \sqsubseteq b \\ env \sqsubseteq env' \\ \langle env', b\rangle = \mathcal{E}[\![e_1]\!]\langle env', b\rangle \end{cases} \\
&\ \textbf{in} \\
&\ \textbf{case } b \textbf{ of} \\
&\ \qquad \bot\!: \langle env', a\rangle \\
&\ \qquad true : \mathcal{E}[\![e_2]\!]\langle env', a\rangle \\
&\ \qquad false : \mathcal{E}[\![e_3]\!]\langle env', a\rangle \\
&\ \qquad otherwise : \top \\
&\ \textbf{endcase} \\
\mathcal{E}[\![x = e_1 in\ e_2]\!]\langle env, a\rangle &= \textbf{lcs} \begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r \\ env' = \mathcal{C}[\![x = e_1]\!] \quad env' \\ \langle env', r\rangle = \mathcal{E}[\![e_2]\!]\langle env', r\rangle \end{cases} \\
&\ \textbf{in } \langle env', r\rangle
\end{aligned}
$$

Figure 2.10: Denotational Semantics of Cid: Expressions

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

Figure 2.11: Denotational Semantics of Cid: Function Symbols

To show that one-step reduction preserves meaning, we need to associate meanings with the basic entities used in the operational semantics, i.e. with configurations. In the following, the semantic function $\mathcal{M}$ assigns to configurations a closure operator over the domain $ENV \times_{\top} V$.

$$
\begin{aligned}
\mathcal{M}[\![\langle D, e, \rho, FL\rangle]\!]\langle env, a\rangle &= \text{lcs}\begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r \\ env' = \mathcal{C}[\![D \cup \rho]\!]\ \ env' \\ \langle env', r\rangle = \mathcal{E}[\![e]\!]\langle env', r\rangle \end{cases} \\
&\quad \text{in } \langle env', r\rangle
\end{aligned}
$$

The function $\mathcal{M}$ is intended to represent the effect of the complete computation on a configuration. The theorem we will prove shows that as we rewrite a configuration the meaning as given by $\mathcal{M}$ will not alter. Since $\mathcal{M}$ assigns a closure operator to an operational configuration, this is equivalent to saying that the set of fixed points of the closure operator assigned to an operational configuration is preserved under reduction of the configuration.

The Adequacy Theorem

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

We define a relationship $\preceq$ between syntactic expressions, e, and closure operators, f, on $ENV \times_{\top} V$. The main theorem proves that for all syntactic expressions e, $\mathcal{E}[\![e]\!] \preceq e$. Intuitively, $\mathcal{E}[\![e]\!] \preceq e$ means that given any finite approximant to the result predicted by $\mathcal{E}[\![e]\!]$, there is a finite sequence of reductions evaluating e in a suitable syntactic environment, that produces a more refined value. In particular, if the result predicted by $\mathcal{E}[\![e]\!]$ is $\top$, evaluating e in a suitable syntactic environment results in *error*. The proof that $\mathcal{E}[\![e]\!] \preceq e$, for all expressions e proceeds by structural induction on the expressions. The inductive argument is an interleaving lemma that constructs the reduction sequence corresponding to a set of semantic equations, assuming that we can construct the reductions corresponding to each equation. The difficult part of any adequacy proof is that one has to construct a reduction sequence from semantic information. In our case, we use the special properties of fixed points of closure operators to carry out this construction. Consider the case is when one has parallel imposition of constraints. We make use of the fact that the semantic prescription for determining the least common fixed point of a pair of closure operators suggests an interleaving of the reduction sequences of the subterms. This is the key to the whole adequacy proof.

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

Full abstraction

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

Definition 10 $e_1 \sqsubseteq_{op} e_2$ *if for all definition contexts* $D[]$ *and for all expression contexts* $C[]$,

- $< D[e_1], C[e_1], \emptyset, FL > \rightarrow b$, *where* b *is a integer implies*
  $< D[e_2], C[e_2], \emptyset, FL > \rightarrow b$ *or* $< D[e_2], C[e_2], \emptyset, FL > \rightarrow error$.
- $< D[e_1], C[e_1], \emptyset, FL > \rightarrow error$ *implies*
  $< D[e_2], C[e_2], \emptyset, FL > \rightarrow error$

The basic results are that the approximation relation between the meanings of terms in the domain accurately reflects the operational preorder. Thus, we prove that $\mathcal{E}[\![e_1]\!] \sqsubseteq \mathcal{E}[\![e_2]\!] \iff e_1 \sqsubseteq_{op} e_2$.

One-step Reduction Preserves Meaning

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

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

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

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

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

In order to show that one-step reduction preserves meaning we need to associate meanings with the basic entities used in the operational semantics, i.e. with configurations. In the following the semantic function $\mathcal{M}$ assigns to configurations a closure operator over the domain $ENV \times_{\top} V$. We use the semantic functions $\mathcal{E}$, $\mathcal{F}$ and $\mathcal{C}$ defined previously and the same notational conventions.

$$\mathcal{M}[\![\langle D, e, \rho, FL\rangle]\!]\langle env, a\rangle \;=\; \text{lcs}\begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r \\ env' = \mathcal{C}[\![D \cup \rho]\!]\;\; env' \\ \langle env', r\rangle = \mathcal{E}[\![e]\!]\langle env', r\rangle \end{cases} \text{in } \langle env', r\rangle$$

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

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

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

Theorem 1 *Suppose that the following rewrite is possible:*

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

then $\forall$ *env satisfying condition* (I) *with respect to both* ρ *and* ρ',

$$(\forall\, a \in V)\ [(\mathcal{M}[\![\langle D, x, \rho, FL\rangle]\!]\langle env, a\rangle)_{bv(\rho)} = \langle env, a\rangle_{bv(\rho)}]$$

if and only if, $(\mathcal{M}[\![\langle D', e', \rho', FL'\rangle]\!]\langle env, a\rangle)_{bv(\rho)} = \langle env, a\rangle_{bv(\rho)}$

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

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

Below, to reduce notational complexity, we prove the result for the case when the context $C[]$ is the simple ccontext $C[] = []$. The result for more complicated contexts follows by a similar proof. Using the definition of $\mathcal{M}$ we get:

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

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

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

Let $\rho(x) = v$. Note that this means that $x = v \in EQ(\rho)$. Thus, we need to prove that the above is equivalent to

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

This reduces to proving that

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

and

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

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

$$\text{lcs}\begin{cases} env \sqsubseteq env' \\ \langle env', r\rangle = \mathcal{E}[\![e]\!]\langle env', r\rangle \\ env'[x] = r \end{cases}$$
$$\text{in } env'$$

Note that

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

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

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

The next case we need to consider are the array operations. The reduction rules are given in Figure 2.3. The proof for the fourth rule is straightforward and is omitted. The proofs for the first and third rule are similar; we present below the proofs for the second and third rules. The second rewrite rule is applicable when the expression e in $array(e)$ has reduced to an integer. The meanings of the two configurations are

$$\mathcal{M}[\![\langle D, array(n), \rho, FL\rangle]\!]\langle env, a\rangle \;=\; \text{lcs}\begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r' \\ env' = \mathcal{C}[\![D \cup \rho]\!] \;\; env' \\ r' = \mathcal{A}rray(K(n)) \sqcup a \end{cases}$$
$$\text{in } \langle env', r'\rangle$$

$$\mathcal{M}[\![\langle D, array(n), \rho, FL\rangle]\!]\langle env, a\rangle \;=\; \text{lcs}\begin{cases} env \sqsubseteq env'' \\ a \sqsubseteq r'' \\ env'' = \mathcal{C}[\![D \cup \rho]\!] \;\; env'' \\ env''[L1] = r'[1] \\ \ldots \\ env''[Ln] = r'[n] \end{cases}$$

in $\langle env'', r''\rangle$

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

The third rewrite rule is applicable for expression of type $e_1\ op\ e_2$. The rule spawns off two processes to compute the sub-expressions e_1 and e_2. The denotations of the configurations before the rewrite is:

$$\mathcal{M}[\![\langle D, e_1\ op\ e_2, \rho, FL\rangle]\!]\langle env, a\rangle = \text{lcs}\left\{\begin{array}{c} env \sqsubseteq env' \\ a \sqsubseteq r' \\ env' = \mathcal{C}[\![D \cup \rho]\!]\ env' \\ \langle env', v_1'\rangle = \mathcal{E}[\![e_1]\!]\langle env', v_1'\rangle \\ \langle env', v_2'\rangle = \mathcal{E}[\![e_2]\!]\langle env', v_2'\rangle \\ r' = (v_1'\ op\ v_2') \sqcup a \end{array}\right.$$
$$\text{in } \langle env', r'\rangle$$

The meaning of the configuration after the rewrite is

$$\mathcal{M}[\![\langle D, x_1\ op\ x_2, \rho, FL\rangle]\!]\langle env, a\rangle = \text{lcs}\left\{\begin{array}{c} env \sqsubseteq env'' \\ a \sqsubseteq r'' \\ env'' = \mathcal{C}[\![D \cup \rho]\!]\ env'' \\ env''[x_1] = v_1 \\ \langle env'', v_1''\rangle = \mathcal{E}[\![e_1^*]\!]\langle env'', v_1''\rangle \\ env''[x_2] = v_2 \\ \langle env'', v_2''\rangle = \mathcal{E}[\![e_2]\!]\langle env'', v_2''\rangle \\ r'' = (v_1''\ op\ v_2'') \sqcup a \end{array}\right.$$
$$\text{in } \langle env'', r''\rangle$$

The two systems of equations are identical except for the introduction of variables x_1, x_2. Since these variables are new and not bound in ρ, result follows.

The final case that we look at is function application. The operational rules are in Figure 2.2. For rule 1, the meanings of the configurations are given by

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

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

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

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

are identical. We need to show that all solutions of 8 and 9 coincide with solutions of 3,4 and 5 and vice-versa. Equation 8 contains all the equations implicit in 3 as well as the new ones obtained by adding the definitions in F to the configuration. The constraint on the argument to the function contained in equation 4 is contained in equation 9. The two systems of equations express the same constraints, thus $\mathcal{M}$ assigns the same meanings to the two configurations.

As far as rule 2 is concerned, the meanings of the configurations are given by :

$$\mathcal{M}[\![\langle D, x = e \textbf{ in } e_1, \rho, FL\rangle]\!]\langle env, a\rangle = \text{lcs}\left\{\begin{array}{c} env \sqsubseteq env' \\ a \sqsubseteq r' \\ env' = \mathcal{C}[\![D \cup \rho]\!]\ env' \\ \langle env', v'\rangle = \mathcal{E}[\![x = e \textbf{ in } e_1]\!]\langle env', v'\rangle \end{array}\right.$$

$$\text{in } \langle env', r' \rangle$$

$$\mathcal{M}[\![\langle D, x = e \textbf{ in } e_1, \rho, FL\rangle]\!]\langle env, a\rangle = \text{lcs}\begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r' \\ env' = \mathcal{C}[\![D \cup \rho \cup \{x = e\}]\!]\ env' \\ \langle env', v'\rangle = \mathcal{E}[\![e_1]\!]\langle env', v'\rangle \end{cases}$$

$$\text{in } \langle env', r' \rangle$$

Proof follows by noticing that $\langle env', v'\rangle = \mathcal{E}[\![x = e \text{ in } e_1]\!]\langle env', v'\rangle$ is equivalent to the set of equations :

$$\begin{aligned} \langle env', v'\rangle &= \mathcal{E}[\![x = e \text{ in } e_1]\!]\langle env', v'\rangle \\ env' &= \mathcal{C}[\![x = e]\!]\ env' \end{aligned}$$

The final issue that we need to address is the soundness of the rewrite rules that use unification to incorporate new identifiers into the collection of alias sets i.e to show that the cases labeled "definitions" in Figure 2.4 preserve the meanings of configurations. The reasoning is quite straightforward. There are two sub-cases to consider, corresponding to the two operational rules for definitions. The first case follows immediately from the inductive hypothesis. In the second case, the equations added to $EQ(\rho)$ are already present in D; thus it was present as a constraint in computing the meaning of the configuration. Similarly, if there is an inconsistency introduced by the unification process then there were inconsistent constraints present in computing the meaning of the original configuration. ∎

Adequacy of Denotational Semantics

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

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

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

Operational facts

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

- $\dfrac{< D,\ e\ , \rho, FL > \rightarrow < D',\ e'\ , \rho', FL' >}{< D,\ e\ , \rho, FL > \longmapsto_s < D',\ e'\ , \rho', FL' >}$
- $< D,\ e\ , \rho, FL > \longmapsto_s < D \bigcup \{x = e'\},\ e\ , \rho, FL >$

$\overset{*}{\longmapsto}_s$ is the reflexive and transitive closure of $\longmapsto_s$.

The transition systems $\rightarrow$ and $\longmapsto_s$ are closely related. The following lemma generalizes the Church-Rosser property, given by lemma 1. It can be viewed as saying that addition of new constraints cannot disable enabled reductions.

Lemma 6 *Let* $conf \overset{*}{\longmapsto}_s conf_1 \wedge conf \overset{*}{\rightarrow} conf_2$. *Then, there exists a configuration* $conf_3$ *such that* $conf_2 \overset{*}{\longmapsto}_s conf_3 \wedge conf_1 \overset{*}{\rightarrow} conf_3$.

Proof for Composition of Definitions

Inclusive predicates are key components in many adequacy proofs [47] [66]. They relate the semantic values with syntactic expressions. They are primarily used to establish that a predicted semantic value is actually attained by rewriting. For defining the inclusive predicate relating expressions and closure operators on $ENV \times_{\top} V$, we need to develop notation that relates syntactic and semantic values as well as syntactic and syntactic environments.

The following definition relates syntactic expressions and semantic values, and syntactic environments and semantic environments. Intuitively, $v \preceq (e, \rho)$ means that that e when evaluated in syntactic environment ρ gives a value that is no less defined than v. $env \preceq \rho$ can be viewed as saying that the syntactic environment ρ is more constrained than env. The third case of the definition combines the first two cases in a natural way. It relates pairs of syntactic expressions and syntactic environments $\langle \rho, e \rangle$ and pairs of semantic values and semantic environments $r = \langle env, v \rangle$.

Definition 11 *Syntactic values and environments are related to semantic values and environments as follows:*

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

Note that an inconsistent environment ρ is defined to dominate all semantic environments env. Furthermore, if $env = env_{\top}$, and $env \preceq \rho$, then ρ is inconsistent. Frequently, we will use $\vec{v} \preceq (\vec{x}, \rho)$ as shorthand for the conjunction $v_1 \preceq (x_1, \rho) \wedge \ldots v_n \preceq (x_n, \rho)$. The associaion of variable names and values will be clear from the context.

The following lemma states that the relation $\preceq$ defined above is *inclusive* [66], and satisfies natural monotonicity properties. The proof is immediate and is omitted.

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

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

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

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

Definition 12 E covers g, *written* $g \preceq E$, *is defined as follows. Let*

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

Then, given $< E,\ \#\ ,\star,\star\star > \stackrel{*}{\longmapsto}_s < D,\ \#\ ,\rho, FL >$,

$$(\forall env_f \sqsubseteq\ env')\ (\exists)\ [< D,\ \#\ ,\rho, FL > \stackrel{*}{\rightarrow} < D',\ \#\ ,\rho_{res}, FL' > \wedge\ env_f \preceq \rho_{res}]$$

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

Now, we prove the case of structural induction corresponding to the case of combining equations. The difficult constituent of an adequacy proof is that one has to construct a reduction sequence from semantic information. In our case, we use the special properties of fixed points of closure operators to carry out this construction. In some sense, this is the key to the whole adequacy proof. Suppose that g_1 and g_2 are two closure operators that correspond to the imposition of two constraints given as sets off equations E_1 and E_2. Suppose that we know how to construct reduction sequences corresponding to E_1 and E_2 individually. Then, since we know that the least common fixed point of g_1 and g_2 is the least fixed point of $(g_1 \circ g_2)$, we can construct an interleaved reduction sequence of E_1 and E_2 corresponding to the computing the iterates of $(g_1 \circ g_2)$. In other words, the special form of the fixed point iteration provides guidance about how to construct the interleaved reduction sequence. The proof of the following lemma formalizes this intuition.

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

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

Proof: Let $E = E_1 \bigcup E_2$. Let

- $g\ env = env'$
- $env \preceq \rho$
- $< E,\ \#\ , \star, \star\star > \overset{*}{\longmapsto}_s < D,\ \#\ , \rho, FL >$

$env' = \bigsqcup_i\{(g_1 \circ g_2)^i\ env|i\}$. So, if $env_f \sqsubseteq env'$, there is an i: $env_f \sqsubseteq (g_1 \circ g_2)^i\ env$. By induction on i, we prove that $env_f \sqsubseteq (g_1 \circ g_2)^i\ env$ implies that there is a reduction sequence, $< D,\ \#\ , \rho, FL > \xrightarrow{*} < D',\ \#\ , \rho_{res}, FL' >$, such that $env_f \preceq \rho_{res}$.

Base: $(i = 0)$

In this case, $env_f \sqsubseteq env$ and the configuration $< D,\ \#\ , \rho, FL >$ 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,

- $env_f \sqsubseteq g_1\ env_1$
- $env_1 \sqsubseteq g_2\ env_2$
- $env_2 \sqsubseteq (g_1 \circ g_2)^i env$

From induction hypothesis, there is a configuration $< D_2,\ \#\ , \rho_2, FL_2 >]$ such that

$$env_2 \preceq \rho_2 \wedge\ < D,\ \#\ , \rho, FL > \xrightarrow{*} < D_2,\ \#\ , \rho_2, FL_2 >$$

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

From hypothesis, $g_2 \preceq E_2$ and $< E_2,\ \#\ , \star, \star\star > \stackrel{*}{\longmapsto}_s < D_2,\ \#\ , \rho_2, FL_2 >$, there is a reduction sequence from $< D_2,\ \#\ , \rho_2, FL_2 > < D_1,\ \#\ , \rho_1, FL_1 >$, such that $env_1 \preceq \rho_1$.

Similarly, since $< E_1,\ \#\ , \star, \star\star > \stackrel{*}{\longmapsto}_s < D_1,\ \#\ , \rho_1, FL_1 >$ and $g_1 \preceq E_1$, there is a reduction sequence from $< D_1,\ \#\ , \rho_1, FL_1 >$ to $< D',\ \#\ , \rho_{res}, FL' >$, such that $env_f \preceq \rho_{res}$. Hence, the result ∎

Full proof

This subsection is devoted to proving that $\mathcal{E}[\![e]\!] \preceq e$, for all expressions e. This section generalizes the ideas contained in the case of structural induction treated above. This section is organized as follows. First, we define the inclusive predicates for relating expressions, function symbols and operator symbols to closure operators of appropriate type. Then, we present the proof of the interleaving lemma. This proof generalizes the ideas contained in the proof of lemma 8 to arbitrary sets of semantic equations. The reminder of the proof proceeds by structural induction on the formation of expressions. The proof is a sequence of lemmas: each lemma being a straightforward reduction to the interleaving lemma. Finally, the proof is done for function definitions. The tools developed previously are used to reduce the proof to a routine fixed point induction [39].

First, we define a relationship between expressions e and closure operators f on $ENV \times_{\top} V$. Roughly speaking, $f \preceq e$ means that when e is evaluated in a suitable syntactic configuration, the resulting value has a meaning that dominates the result predicted by f. This is done in two stages. First, we define a relationship $(x, e) \models f \preceq e$. $(x, e) \models f \preceq e$ means that when e is evaluated in a suitable syntactic configuration with the variable x bound to e, the resulting value has a meaning that dominates the result predicted by f. We use this idea to define $f \preceq e$.

Definition 13 $\{(x, e)\} \models f \preceq \{x = e\}$ *is defined as follows. Let*

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

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

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

Now, we have all the tools to define a relationship between expressions e and closure operators f on $ENV \times_{\top} V$.

Definition 14 $f \preceq e \Leftrightarrow (\forall x)\ [\{(x, e)\} \models f \preceq \{(x = e)\}]$.

The special case of the above definition that we are interested in is when $a_f = \perp$, and $< \{x = e\},\ x\ , \star, \star\star > \ = \ < D,\ x\ , \rho, FL >$, and $f = \mathcal{E}[\![e]\!]$. As before, the greater generality of the definition simplifies the proofs. Let ρ be more constrained than env. Let $\mathcal{E}[\![e]\!]\langle env, \perp\rangle = \langle env', b\rangle$. Then, given any finite approximant $\langle env_f, b_f\rangle$ to $\langle env', b\rangle$, there is a finite reduction sequence evaluating expression e in syntactic environment ρ such that the resulting syntactic environment ρ_{res} is more constrained than env_f, and the resulting expression e' evaluated in ρ_{res} yields a more defined value than b_f. In particular, if env' is the error environment, evaluating e in ρ results in $error$.

The above two definitions can definition can be combined and generalized to sets of equations and expressions as follows. In the following definition, we use $(\vec{x}, \vec{e})$ as shorthand for $\{(x_1, e_1) \ldots (x_n, e_n)\}$.

Definition 15 *Let f be a closure operator on $ENV \times_{\top} \underbrace{V \times_{\top} V \ldots}_{n}$. Then, $E \cup \{(\vec{x}, \vec{e})\} \models f \preceq E \cup \{(\vec{x}, \vec{e})\}$ is defined as follows. Let*

- $f\langle env, \langle a_{f1} \ldots a_{fn}\rangle\rangle = \langle env', \langle r_{f1} \ldots r_{fn}\rangle\rangle$
- $env \preceq \rho$
- $(\forall 1 \leq i \leq n)\ [a_{fi} \preceq (x_i, \rho)]$

Then, given $< E \cup \{x_1 = e_1 \ldots x_n = e_n\},\ \#\ , \star, \star\star > \overset{*}{\longmapsto}_s < D,\ \#\ , \rho, FL >$, for all finite approximants $\langle env_f, \langle r_{f1} \ldots r_{fn}\rangle\rangle$ to $\langle env', \langle r_1 \ldots r_n\rangle\rangle$, we have

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

The following lemma is the analogue of lemmas 7. The proof is immediate and is omitted.

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

1. $g \preceq E \wedge g' \sqsubseteq g \Rightarrow g' \preceq E$
2. *Let $\{g_i | i\}$ be a chain in the space of closure operators on ENV. Then,*

$$(\forall i)\ [g_i \preceq E] \Rightarrow \bigsqcup_i \{g_i | i\} \preceq E$$

3. $f \preceq e \wedge f' \sqsubseteq f \Rightarrow f' \preceq e$
4. *Let $\{f_i | i\}$ be a chain in the space of closure operators on $ENV \times_{\top} V$. Then, $(\forall i)\ [f_i \preceq e]$ implies $\bigsqcup_i \{f_i | i\} \preceq e$*

Recall that the denotation of the function symbol F was a closure operator on $V \times_{\top} V$. The following definition relates closure operators on $V \times_{\top} V$ to the function symbol F.

Definition 16 $\{(x_1 = F(x_2))\} \models s \preceq \{(x_1 = F(x_2))\}$ *is defined as follows. Let*

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

Then, given $< \{x_2 = F(x_1)\},\ \#\ , \star, \star\star > \stackrel{*}{\longmapsto}_s < D,\ \#\ , \rho, FL >$, *and any* $\vec{b}_f \sqsubseteq \vec{b}$,

$$(\exists)[< D,\ \#\ , \rho, FL > \stackrel{*}{\rightarrow} < D',\ \#\ , \rho_{res}, FL' >) \wedge \vec{b}_f \preceq \langle \rho_{res}, \vec{x}\rangle]$$

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

Definition 17 $s \preceq F \Leftrightarrow (\forall x_1, x_2)\ [\{(x_1 = F(x_2))\} \models s \preceq \{(x_1 = F(x_2))\}]$

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

Definition 18 $\{(y = op(x_1 \ldots x_n)\} \models t \preceq \{(y = op(x_1 \ldots x_n)\}$ *is defined as follows. Let* $< \{y = op(x_1, \ldots x_n)\},\ \#\ , \star, \star\star > \stackrel{*}{\longmapsto}_s < D,\ \#\ , \rho, FL >$ *and,*

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

Then, $\forall \vec{b_f} \sqsubseteq \vec{b}$, $(\exists)\ [< D,\ \#\ , \rho, FL > \stackrel{*}{\rightarrow} < D',\ \#\ , \rho_{res}, FL' >]$ *such that*

$$b_{(n+1)f} \preceq \langle \rho_{res}, y\rangle \wedge (\forall 1 \leq i \leq n)\ [b_{if} \preceq \langle \rho_{res}, x_i\rangle]$$

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

Definition 19 $s \preceq op \Leftrightarrow (\forall y, \vec{x})\ [\{(y = op(\vec{x})\} \models t \preceq \{(y = op(\vec{x})\}]$

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

Definition 20 *Let op be an n-ary operator, n a positive integer. op is* safe, *if for any configuration* $< D,\ e\ , \rho, FL >$*,we have the following.*

$$(\forall 1 \leq i \leq n)\ [v_i \preceq (x_i, \rho)] \Rightarrow [op(v_1, \ldots v_n) \preceq (op(x_1 \ldots x_n), \rho)]$$

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

Interleaving lemma

The aim of this subsection is to prove that the $\preceq$ relation is respected by parallel composition. Recall that lemma 8 worked only for the specific case of parallel composition of constraints. The interleaving lemma generalizes the statement and the proof to work for any collection of syntactic objects. Recall that all the cases of the semantics were described in terms of solving sets of equations. The following lemma works for all the forms of equations used in the semantics. The idea behind the lemma is as follows. Suppose that we are given a list of syntactic entities and their associated closure operators. Assume that the syntactic objects "dominate" the corresponding semantic entities. The lemma below shows that the semantic parallel composition of the given closure operators is dominated by the syntactic parallel composition of the given syntactic entities. As in lemma 8, the special form of the fixed point iteration provides guidance about how to construct the interleaved reduction sequence.

The attempt to describe the structure of a general enough system of equations is the primary cause for the notational complexity of the statement of the Interleaving lemma. Consequently, we start off by describing the notation of the statement of the lemma.

Constraints: The syntactic entities are definitions $E_1 \dots E_m$ and the semantic entities are closure operators $g_1, \dots g_m$ on ENV. These are respectively denoted by E_i and g_i, with $g_i \preceq E_i$. Let $S_0 = \cup_i E_i$.

Expressions: The syntactic entities are expressions $e_1 \dots e_n$ and their semantic counterparts are closure operators $f_1, \dots f_n$ on $ENV \times_{\top} V$, with $\{(x_i, e_i)\} \models f_i \preceq \{(x_i, e_i)\}$. Let $S_1 = \{(x_i = e_i) | i\}$.

Functions: The syntactic entity is the function symbol F and its semantic counterpart are closure operators $s_1 \dots s_p$ on $V \times_{\top} V$, with $\{(x_{n+2i-1} = F(x_{n+2i}))\} \models s_i \preceq \{(x_{n+2i-1} = F(x_{n+2i}))\}$. Let $S_2 = \{(x_{n+2i-1} = F(x_{n+2i})) | i\}$.

Binary operators: The syntactic entities are binary operators $op^{21} \dots op^{2r_2}$ and semantic counterparts are closure operators $t^{21} \dots t^{2r_2}$ on $V \times_{\top} V \times_{\top} V$, $\{(x_{(n+2p+3i-2)} = op^{2i}(x_{(n+2p+3i-1)}, x_{(n+2p+3i)}))\} \models t^{2i} \preceq \{(x_{(n+2p+3i-2)} =$

$op^{2i}(x_{n+2p+3i-1}, x_{n+2p+3i})\}$. Following previous notation, let $S_3 = \{x_{n+2p+3i-2} = op^{2i}(x_{n+2p+3i-1}, x_{n+2p+3i})|i\}$.

Unary operators: The syntactic entities are unary operators $op^{11} \dots op^{1r_1}$, their models are closure operators $t^{11} \dots t^{1r_1}$ on $V \times_{\top} V$, with $\{x_{n+2p+3r_2+2i-1} = op^{1i}(x_{n+2p+3r_2+2i}\} \models t^{1i} \preceq \{(x_{(n+2p+3r_2+2i-1)} = op^{1i}(x_{(n+2p+3r_2+2i)})\}$. Let $S_4 = \{(x_{(n+2p+3*_2+2i-1)} = op^{1i}(x_{(n+2p+3r_2+2i)})|i\}$.

Aliasing: In the statement of the lemma, "semantic aliasing" is made explicit in the system of semantic equations called *Semeq*. The operational analogue is the system of syntactic equations *Syneq*. For example, consider the semantic definition of application:

$$\mathcal{E}[\![F(e)]\!]\langle env, a\rangle \quad = \quad \text{lcs} \begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r \\ \langle env', v\rangle = \mathcal{E}[\![e]\!]\langle env', v\rangle \\ \langle v, r\rangle = \mathcal{E}_F[\![F]\!]\langle v, r\rangle \end{cases} \text{in } \langle env', r\rangle$$

The "semantic aliasing" is implicit in the use of the same name v in two of the equations. In the statement of the lemma below, we are going to make this explicit. In particular, the above would be rewritten as:

$$\mathcal{E}[\![F(e)]\!]\langle env, a\rangle \quad = \quad \text{lcs} \begin{cases} env \sqsubseteq env' \\ a \sqsubseteq r \\ \langle env', v_1\rangle = \mathcal{E}[\![e]\!]\langle env', v_1\rangle \\ \langle v_2, r\rangle = \mathcal{E}_F[\![F]\!]\langle v_2, r\rangle \\ v_1 = v_2 \end{cases} \text{in } \langle env', r\rangle$$

The "semantic aliasing" is made explicit by the equation $v_1 = v_2$.

Lemma 10 INTERLEAVING LEMMA:

Let h be the closure operator on $ENV \times_{\top} \underbrace{V \times_{\top} V \ldots}_{n+2p+3r_2+2r_1}$ defined as: $h\langle env, \vec{a}\rangle =$

$$\textbf{lcs} \begin{cases} env \sqsubseteq env' \\ \vec{a} \sqsubseteq \vec{a}' \\ env' = g_i\ env', i = 1 \ldots m \\ \langle env', a'_i\rangle = f_i\langle env', a'_i\rangle,\ i = 1 \ldots n \\ \langle a'_i, a'_{i+1}\rangle = s_k\langle a'_i, a'_{i+1}\rangle,\ i - n = 1 \ldots 2p - 1 \\ \langle a'_i, a'_{i+1}, a'_{i+2}\rangle = t^{2(i-n+2p)}\langle a'_i, a'_{i+1}, a'_{i+2}\rangle,\ i - n - 2p = 1 \ldots 3r_2 - 2 \\ \langle a'_i, a'_{i+1}\rangle = t^{1(i-n-2p)}\langle a'_i, a'_{i+1}\rangle,\ i - n - 2p - 3r_2 = 1 \ldots 2r_1 - 1 \\ Semeq \end{cases}$$

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

where Semeq is a set of equalities of the form $w_1 = w_2$, where $w_1, w_2 \in \{a'_1 \ldots a'_{n+2p+3r+2r_1}\}$. Let Syneq be a list of equations of form determined by Semeq, with a equation $\{x_i = x_j\}$ for every $a'_i = a'_j$ in Semeq. Then, $\cup_i S_i \models h \preceq \cup_i S_i \cup Syneq$.

Proof: Rather than doing the proof for the general case that involves multiple indices, we sketch the special case of the proof that resembles the semantic definition for an expression of form $e_1\ op\ e_2$. The sketch is in sufficient detail to bring out the resemblance to the proof of lemma 8. In particular, $m = 0, n = 2, p = 0, r_1 = 0, r_2 = 1$. The definition of h now takes the form:

$$h\langle env, \vec{a}\rangle \;=\; \text{lcs} \begin{cases} \vec{a} \sqsubseteq \vec{b} \\ env \sqsubseteq env' \\ \langle env', b_1\rangle = f_1\langle env', b_1\rangle \\ \langle env', b_2\rangle = f_2\langle env', b_2\rangle \\ \langle b_3, b_4, b_5\rangle = t\langle b_3, b_4, b_5\rangle \\ b_1 = b_4 \\ b_2 = b_5 \end{cases}$$

$$\text{in } \langle env', \vec{b}\rangle$$

First, we alter the definitions of the given closure operators so that their types match the type of h. Thus, the domain of definition of each closure operator is made to be

$ENV \times_\top \underbrace{V \times_\top V \ldots}_{5}$. This is done in the natural way. The new closure operators only change some of the V components and leave the others untouched. It will turn out that h can be recovered as the least upper bound of the closure operators so defined. More precisely, define closure operators $h'_1, \ldots h'_4$, as follows:

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

Let $u = (h'_1 \circ \ldots h'_4)$. Note that $h = \bigsqcup_l \{u^l | l = 1, 2 \ldots\}$. So, from lemma 9, it suffices to prove $(\forall l)$ $[\{x_1 = e_1, x_2 = e_2, x_3 = op(x_1, x_2)\} \models u^l \preceq_{\vec{x}} conf]$, where $conf$ is $\{x_1 = x_4, x_2 = x_5, x_1 = e_1, x_2 = e_2, x_3 = op(x_1, x_2)\}$.

This is done by induction on l. This proof follows closely the proof of lemma 8 and is omitted. ∎

Cases of Structural Induction

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

Lemma 11 *(Cases of structural induction):*

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

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

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

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

$$\mathbf{in}\langle env', b\rangle$$

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

$$h\ \langle env, a\rangle = \mathbf{lcs}\begin{cases} \langle env, a\rangle \sqsubseteq \langle env', b\rangle \\ \langle env', b\rangle = f\langle env', b\rangle \\ env' = g\ env' \end{cases}$$
$$\mathbf{in}\langle env', b\rangle$$

Array creation: $f \preceq e \Rightarrow g \preceq array(e)$, *with* g *a closure operator on* $ENV \times_{\top} V$ *defined as*

$$g\langle env, a\rangle = \mathbf{lcs}\begin{cases} env \sqsubseteq env' \\ \langle env', n\rangle = f_1\langle env', n\rangle \\ r = Array(n) \sqcup a \end{cases}$$
$$\mathbf{in}\langle env', r\rangle$$

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

$$h\langle env, a\rangle = \mathbf{lcs}\begin{cases} env \sqsubseteq env' \\ \langle env', v\rangle = f\langle env', v\rangle \\ \langle v, r\rangle = s\langle v, r\rangle \end{cases}$$
$$\mathbf{in}\langle env', r\rangle$$

Conditional expressions: *Let* $f_i \preceq e_1$, *i=1,2 3. Then,* $g \preceq cond(e_1, e_2, e_3)$ *where* g *is the closure operator on* $ENV \times_{\top} V$ *defined as follows:*

$$\begin{aligned} g\langle env, a\rangle = {} & \text{let } f_1\langle env, \perp\rangle = \langle env', bool\rangle \text{ in} \\ & \textbf{if } bool\ then\ f_2\langle env', a\rangle \\ & \text{else } f_3\langle env', a\rangle \end{aligned}$$

Base case for arrays: $\mathcal{E}[\![[L_1 \ldots L_n]]\!] \preceq [L_1 \ldots L_n]$

Base case for variables: $\mathcal{E}[\![x]\!] \preceq x$

Proof: The proofs are a straightforward application of the interleaving lemma 10. Below, we give the proofs for the first two cases, as examples of the use of the interleaving lemma. The machinery developed so far renders the proofs of the remaining cases routine, and these proofs are omitted.

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

$$g'\ \langle env, \langle a_1, a_2 \rangle\rangle \;=\; \text{lcs} \begin{cases} env \sqsubseteq env' \\ \vec{a} \sqsubseteq \vec{b} \\ \langle env', b_1 \rangle = f\langle env', b_1 \rangle \\ b_2 = env'[x] \\ b_1 = b_2 \end{cases}$$
$$\text{in}\langle env', b \rangle$$

From assumption $f \preceq e$, $\{x = e\} \models f \preceq \{x = e\}$, for all x. From lemma 10, $\{x = e, x = y\} \models g' \preceq \{x = e, x = y\}$, for all y. Thus, choosing $y = x$, we get, $\{x = e, x = x\} \models g' \preceq \{x = e, x = x\}$. Note that $g'\ \langle env, \langle \perp, \perp \rangle\rangle = \langle env', \langle \perp, \perp \rangle\rangle$, where $g\ env = env'$. Thus, $g \preceq x = e$.

2. Consider h' the closure operator on $ENV \times_T \underbrace{V \times_T \ldots V}_{5}$, defined as,

$$h'\ \langle env, \vec{a} \rangle \;=\; \text{lcs} \begin{cases} \langle env, \vec{a} \rangle \sqsubseteq \langle env', \vec{b} \rangle \\ \langle env', b_1 \rangle = f_1\langle env', b_1 \rangle \\ \langle env', b_2 \rangle = f_2\langle env', b_2 \rangle \\ b_3\ op\ b_4 \sqsubseteq b_5 \\ b_1 = b_3 \\ b_2 = b_4 \end{cases}$$
$$\text{in}\langle env', \vec{b} \rangle$$

Let $f\langle env, a_f \rangle = r, env \preceq \rho, a_f \preceq (x, \rho)$. Also asssume that

$$< \{x_3 = e_1\ op\ e_2\},\ \#\ , \star, \star\star > \overset{*}{\longmapsto}_s < D,\ \#\ , \rho, FL >$$

Then, we have

$$< x_3 = e_1\ op\ e_2),\ \#\ , \star, \star' > \longrightarrow < \{x_3 = x_1 op x_2, x_1 = e_1, x_2 = e_2\},\ \#\ , \star, \star' >$$

where x_2, x_3 are in FL. From lemma 6, there is a configuration $conf$ such that $< D,\ \#\ , \rho, FL > \xrightarrow{*} conf$ and $< \{x_3 = x_1\ op\ x_2, x_1 = e_1, x_2 = e_2\},\ \#\ , \star, \star\star > \longmapsto^{*}_{s} conf$. So, without loss of generality, we can assume

$$< \{x_3 = x_1\ op\ x_2, x_1 = e_1, x_2 = e_2\},\ \#\ , \star, \star\star > \longmapsto^{*}_{s} < D,\ \#\ , \rho, FL >$$

From assumptions $f_1 \preceq e_1$, $f_2 \preceq e_2$, $\mathcal{E}[\![op]\!] \preceq op$ and lemma 10, $(\forall x_1, x_2, x_3)$

$$\{x_1 = e_1, x_2 = e_2, x_3 = op(x_1, x_2)\} \models h' \preceq \{x_1 = e_1, x_2 = e_2, x_3 = op(x_1, x_2)\}$$

where x_1, x_2, x_3 are any variable name. Result follows. ∎

Function definition

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

Lemma 12 *$s \preceq F \Rightarrow \tau(s) \preceq F$, where τ is the continuous function on closure operators on $ENV \times_{\top} V$, defined as below.*

$$\begin{aligned} \tau(f) &= \lambda\,(v, a). \\ & \quad \mathrm{lcs} \left\{ \begin{array}{c} \{x \mapsto v, y \mapsto \perp, F \mapsto f\} \sqsubseteq env' \\ a \sqsubseteq r \\ env \sqsubseteq env' \\ \langle env', r\rangle = \mathcal{E}[\![body_F]\!]\langle env', r\rangle \end{array} \right. \\ & \quad \mathbf{in}\ \langle env'[x], r\rangle \end{aligned}$$

Recall that $\mathcal{F}[\![F]\!]$ was defined in Section 2.1.5 as the least fixpoint of τ.

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

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

The above can be rewritten as

$$\tau(s)(f)\ env\ a \;=\; \text{lcs}\begin{cases} env \sqsubseteq env' \\ \{x \mapsto v, y \mapsto \perp, F \mapsto s\} \sqsubseteq env'' \\ a \sqsubseteq r \\ \langle env', v\rangle = f\langle env', v\rangle \\ \langle env'', r\rangle = \mathcal{E}[\![body_F]\!]\langle env'', r\rangle \\ env''[x] \sqsubseteq env'[x] \end{cases}$$
$$\text{in } \langle env', r\rangle$$

From inductive hypothesis $s \preceq F$ and using lemma 11, we have $\mathcal{E}[\![body_F]\!] \preceq body_F$. Another use of lemma 11 allows us to deduce the desired result, $\tau(s)(f) \preceq F$. ∎

Lemma 13 $\mathcal{F}[\![F]\!] \preceq F$.

Proof: It is easy to check that $\perp \preceq F$, where $\perp$ is the identity function on $V \times_T V$, the least closure operator on $V \times_T V$. From lemma 12, we deduce $(\forall k)\ [\tau^k(\perp) \preceq F]$. As an analogue of lemmas 7, 9 which prove the inclusivity of various forms of the predicate $\preceq$, we have $(\forall k)\ [\tau^k(\perp) \preceq F] \Rightarrow \mathcal{F}[\![F]\!] \preceq F$. Thus $\mathcal{F}[\![F]\!] \preceq F$ ∎

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

Theorem 2 *For all expressions e, $\mathcal{E}[\![e]\!] \preceq e$*

Proof: Lemma 11 proves the base case, the structural induction step. ∎

Full abstraction

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

Definition 21 $e_1 \sqsubseteq_{op} e_2$ *if for all definition contexts $D[]$ and for all expression contexts $C[]$,*

- $< D[e_1], C[e_1], \emptyset, FL > \rightarrow b$, *where b is a integer implies* $< D[e_2], C[e_2], \emptyset, FL > \rightarrow b$ *or* $< D[e_2], C[e_2], \emptyset, FL > \rightarrow error$.

- $< D[e_1], C[e_1], \emptyset, FL > \rightarrow error$ *implies*
  $< D[e_2], C[e_2], \emptyset, FL > \rightarrow error$

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

Theorem 3 *The denotational semantics is inequationally adequate i.e*

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

Proof: Let $\mathcal{E}[\![e_1]\!] \sqsubseteq \mathcal{E}[\![e_2]\!]$ and $< D[e_1], C[e_1], \emptyset, FL > \rightarrow b$. Since one-step reduction preserves meaning, there is an env such that $\mathcal{M}[\![\langle D[e_1], C[e_1, \emptyset, FL\rangle]\!]\langle\perp,\perp\rangle = \langle env, b\rangle$. Since context operations are monotone, $\mathcal{M}[\![\langle D[e_2], C[e_2], \emptyset, FL\rangle]\!]\langle\perp,\perp\rangle$ subsumes $\langle env, b\rangle$.

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

$$\begin{aligned} F(x_1, \ldots x_n) \;\; = & \\ & x_1 = E_1[e_2] \\ & \ldots \\ & x_n = E_n[e_2] \\ & \\ & \text{in } C[e_2] \end{aligned}$$

Note that $< \emptyset, F(x_1, \ldots x_n), \emptyset, FL > \rightarrow < D[e_2], C[e_2], \emptyset, FL >$. From onestep reduction preserves meaning, $\mathcal{E}[\![F(x_1 \ldots x_n)]\!]\langle\perp,\perp\rangle = \mathcal{M}[\![\langle D[e_2], C[e_2, \emptyset, FL\rangle]\!]\langle\perp,\perp\rangle$. Hence, $\langle\perp, b\rangle \sqsubseteq \mathcal{E}[\![F(x_1, \ldots x_n)]\!]\langle\perp,\perp\rangle$. So we have one of the following.

- $< \emptyset, F(x_1, \ldots x_n), \emptyset, FL > \rightarrow b$ or

- $< \emptyset, F(x_1, \ldots x_n), \emptyset, FL > \rightarrow error$

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

- $< D[e_2], C[e_2], \emptyset, FL > \rightarrow b$ or

- $< D[e_2], C[e_2], \emptyset, FL > \rightarrow error$

Hence, the result. ∎

The equivalence of the two orders is full-abstraction. It is essentially a consequence of the act that all the prime elements of the space of the closure operators on $ENV \times_{\top} V$ are expressible as the meanings of expressions. As in Plotkin's proof of full-abstraction for PCF [53], the crux of the proof below is the construction of contexts that can semantically distinguish two different expressions.

Theorem 4 *The denotational semantics is fully-abstract i.e.*

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

Proof: The forward implication was proved in the previous theorem. For the reverse implication consider the case when $\mathcal{C}[\![e_1]\!] \not\sqsubseteq \mathcal{C}[\![e_2]\!]$. From algebraicity of the semantic domains, if $\mathcal{C}[\![e_1]\!] \not\sqsubseteq \mathcal{C}[\![e_2]\!]$, there are finite elements $\langle env_1, v_1 \rangle$, $\langle env_2, v_2 \rangle$ satssifying the following:

$f_{\langle env_1, v_1 \rangle \Rightarrow \langle env_2, v_2 \rangle} \sqsubseteq \mathcal{E}[\![e_1]\!] \wedge f_{\langle env_1, v_2 \rangle \Rightarrow \langle env_2, v_2 \rangle} \not\sqsubseteq \mathcal{E}[\![e_2]\!]$. In this expression, $f_{\langle env_1, v_1 \rangle \Rightarrow \langle env_2, v_2 \rangle}$ is the step function defined as:

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

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

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

$$\begin{aligned} F(x) \quad &= \\ & \quad E \\ & \quad E' \\ & \\ & \quad \text{in } cond(C[x], 0, 1). \end{aligned}$$

We shall prove that $< \emptyset, F([]), \emptyset, FL >$ is the required operational context to distinguish e_1 and e_2. The proof proceeds in two stages:

- First, we deduce that $<\emptyset, F(e_1), \emptyset, FL>$ reduces to 0 or to error. Note that

$$\begin{aligned}\mathcal{E}[\![F(e_1)]\!]\langle\perp,\perp\rangle &= \mathcal{M}[\![\langle\emptyset, F_{e_1}, \emptyset, FL\rangle]\!]\langle\perp,\perp\rangle \\ &= \mathcal{M}[\![\langle G, cond(C[x],0,1), \emptyset, FL\rangle]\!]\langle\perp,\perp\rangle\end{aligned}$$

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

$$\mathcal{E}[\![\langle G, cond(C[x],0,1), \emptyset, FL\rangle]\!] \quad \langle\perp,\perp\rangle = \text{lcs}\begin{cases} \mathcal{C}[\![E]\!]\ env = env & 1 \\ \mathcal{C}[\![x = e_1]\!]\ env = env & 2 \\ \mathcal{C}[\![E']\!]\ env = env & 3 \\ \mathcal{E}[\![C[x]]\!]\langle env, a\rangle = a & 4 \end{cases} \\ \text{in } \langle env, a\rangle$$

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

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

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

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

2.2 Id: the full language

This section is the study of the full language Id. An informal example is used to illustrate the subtlety of the higher order language. This example shows that higher-order functions and logic variables give rise to computational behavior similar to that of own variables in Algol-60 [48]. The operational semantics is presented formally, followed by the abstract semantics. A standard extension of the tools developed in the study of the first order language is then used to show that the denotational semantics is adequate with respect to the operational semantics.

2.2.1 Higher order functions and logic variables

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

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


 {A = array(2);
  g = f A;              ----(5)
  t1 = g 1;             ----(6)
  t2 = g 2;             ----(7)
  in A}
```

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

2.2.2 Syntax

To focus on the essentials, we define a subset of Id. This subset is rich enough that any Id program can be translated in a straight-forward manner into the subset. This is done to reduce the number of cases to be considered for the operational and denotational semantics. For the rest of this chapter, we shall call this subset Id.

Figure 2.12 describes the syntax of Id. As in the first order fragment, the loop construct is eliminated since a loop can be replaced by a tail recursive function. We

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

Figure 2.12: Syntax of Id

also assume that the left-hand side of a definition is an identifier; a definition of the form `e1[e2] = e3` can be replaced by two definitions `x = e1[e2]; x = e3` where `x` is a new identifier. This is reasonable if we think of definitions as constraints. In addition, we will assume that all local variables have been made into parameters so that the body of a function does not introduce any new names.

We assume that the language is simply typed. Thus, for example, an abstraction $\lambda x.$ exp has type $\sigma \rightarrow \tau$ if the variable `x` has type σ and `exp` has type τ. Similarly, an application `exp1(exp2)` is well typed only if `exp1` has type $\sigma \rightarrow \tau$, `exp2` has type σ, for some σ and τ. In this case, the expression `exp1(exp2)` has type τ. Arrays, booleans and integers are considered to be of base type, usually referred to as B. For definitions of the form `x= e`, `x` must have the same type as `e`. In the rest of this chapter, we will eschew the details of typing and assume that the expressions are typed correctly in the usual sense. We will also assume that there is a countable distinct set of variables associated with each type. Finally, we will assume that programs return first order results.

Since we do not perform unification of λ-abstractions, we impose some syntactic restrictions to ensure that there are no multiple definitions of functions. This is done through a function *Defined* that computes the functional variables that are the right hand side of some definition. More formally, we have the following development:

- $Defined(id^{\sigma \rightarrow \tau} = e) = \{id\} \cup Defined(e)$. We require $id \notin Defined(e)$.

- $Defined(id^{B} = exp) = Defined(exp)$.

- $Defined(def; def - list) = Defined(def) \cup Defined(def - list)$. We require $Defined(def) \cap Defined(def - list) = \emptyset$.

- $Defined(x) = Defined(constant) = \emptyset$.

- $Defined(e_1\ op\ e_2) = Defined(e_1(e_2)) = Defined(e_1) \cup Defined(e_2)$. We require $Defined(e_1) \cap Defined(e_2) = \emptyset$.

- $Defined(\lambda x.exp) = Defined(exp)$. If x is a function variable, we require $x \notin Defined(exp)$. Thus, for a higher order variable x, all occurrences of x in exp are in applicative contexts of the form $x(e)$.

- $Defined(def - listin\ exp) = Defined(def - list) \cup Defined(exp)$. We require $Defined(exp) \cap Defined(def - list) = \emptyset$.

2.2.3 Structured Operational semantics of Id

In this section, we give an operational semantics for Id using Plotkin-style [54] state transition rules. Rather than rewrite expressions directly, it is convenient to work with *configurations.* A configuration is a quintuple $< D, e, \rho_F, \rho, FL >$ — intuitively, D contains definitions whose right-hand sides have not yet been completely reduced to an identifier, constant, array, or an abstraction of the form $\lambda x.e$. The expression e in the configuration is the expression whose value is to be produced as the result of the program. Configurations are rewritten by reduction and by constraint solving. For example, an occurrence of $2 + 3$ in D or in e can be replaced by 5 in a reduction step. Once the right-hand side of a definition in D has been reduced completely, the definition can participate in constraint solving. Configurations have two components named ρ_F and ρ which keep track of such definitions. When the right hand side of a definition in D reduces to a λ-abstraction, it is moved into ρ_F, the *function environment.* Since λ-abstractions are not unified, an identifier bound to a λ-abstraction by a definition cannot occur on the left hand side of any other definition; hence, ρ_F is simply a list of identifier/λ-abstraction pairs. The second component, ρ, called the environment, keeps track of bindings between identifiers and *base values* (identifiers, constants and arrays) and has a more complex structure to permit unification — it consists of a (possibly empty) set of *alias-sets* where an alias-set is an equivalence class of base values. For example, $\{x, y, z\}, \{x, y, 4\}$ and $\{x, y, [L1, L2]\}$ are alias-sets. If $b1$ and $b2$ are two base values in the same alias-set, then occurrences of $b1$ in D and e may be replaced by $b2$ without changing the meaning of the program. If unification fails, the configuration is rewritten to 'Error' and computation aborts. Otherwise, the resulting environment replaces the old one in the configuration, and rewriting continues.

We define some syntactic categories required for the operational semantics.

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

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

$D \in$ Defs $::= \phi|$def$_1$, ..., def$_n$ $\qquad$ $e \in$ expression

$A \in$ Alias-set $::= \{B_1, ..., B_n\}$ $\qquad$ $B \in$ Base-value $::= x|c|Ar$

$x, L \in$ Id = set of identifiers $\qquad$ $c \in$ Constant = set of constants

$Ar \in$ Array $::= [x_1, ..., x_n]$ $\qquad$ $\rho \in$ Environment $::= \phi|\{A_1, ..., A_n\}$

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

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

The details of the unification algorithm are identical to the first order fragment. The reader is referred to section 2.1.3. The environmental lookup rule is extended to accommodate higher order variables. In a functional language, an environment is considered to be a function from identifiers to values. In our system, the function environment ρ_F can be interpreted the same way. This leads to the following definition, that is a straightforward extension of the analogous firstorder definition in Section 2.1.3.

Definition 22 *Let $< D, e, \rho_F, \rho, FL >$ be a configuration and x is an identifier not a member of FL. The function $\mathcal{V}(x)$ is defined by cases on the type of x A:*

1. *x is a variable of base type: Let A be the (unique) alias-set that contains x. $\mathcal{V}(x)$ is defined by cases depending on A:*
   - *All the elements of A are identifiers. In this case, $\mathcal{V}(x)$ is undefined.*
   - *At least one element of A is a constant c. Since A is consistent, the elements of A are either identifiers or the constant c. We define $\mathcal{V}(x)$ to be c.*

ilarly, the meaning of the definition $y = 1$ is the closure operator f on ENV defined by $\lambda env.\ env[y \mapsto env(y) \bigsqcup 1]$. Consider the two definitions together: $x = y + 3; y = 1$. Let the meaning of the two definitions together be denoted by h. Let $env_\perp$ be the environment in which all identifiers are undefined. Then, from definition 8, $h\ env_\perp$ is the limit of the sequence $env_\perp, f(g(env_\perp)), f(g(f(g(env_\perp)))), \ldots$. Note that

$$g(env_\perp) = env_\perp$$

$$f(g(env_\perp)) = f(env_\perp) = env_\perp[y \mapsto 1]$$

$$g(f(g(env_\perp))) = g(env_\perp[y \mapsto 1]) = env_\perp[x \mapsto 4, y \mapsto 1]$$

$$f(g(f(g(env_\perp)))) = g(f(g(env_\perp)))$$

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

The next section develops these ideas more formally.

Formal Denotational Semantics of Cid

In defining the denotational semantics, we need environments that assign values in V to identifiers. ENV, the domain of these semantic environments, is $Id \to V$; env, env' will be used to refer to any element of this domain. The environment in which all identifiers are mapped to $\top$ is called $env_\top$. The semantic function $\mathcal{C}$ interprets definitions as closure operators of type $ENV \to ENV$ and the semantic function $\mathcal{E}$ interprets expressions as closure operators of type $(ENV \times_\top V) \to (ENV \times_\top V)$. We assume that the environment contains distinguished bindings for each of the top-level functions permitted in a Cid program. Strictly speaking, the type of ENV must be modified so that an environment is the sum of an ordinary environment as described above and an environment in which the values of functions can be looked up. This fine distinction is not made in the treatment of the first order language.

The semantic clause for function application in Figure 2.11 uses the auxiliary function $\mathcal{E}_F$. This function defines the meaning of functional expressions. We assume that all functions have exactly one argument in order not to clutter up the notation. The function $\mathcal{E}_F$ takes an environment and looks up the definition in the functional part of the environment giving a closure operator on $V \times_\top V$. This function is defined below using a least fixed point operator, written μ, on the space of closure operators on

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

2. *x is a variable of a function type: In this case, $\mathcal{V}(x)$ is L where $x = L$ is the unique definition of x in ρ_F.*

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

The Plotkin-style operational semantics [54] for Id is given in Figures 2.13 and 2.14. The first rule replaces free occurrences of a first order variable x by $\mathcal{V}(\mathrm{x})$ in any context, if $\mathcal{V}(\mathrm{x})$ is defined. Arbitrary contexts are denoted by $C[]$ in this rule. This rule, together with the first rule for definitions, ensures that a free occurrence of an identifier in a configuration can be replaced by the value associated with the identifier in the environment or functional environment. Most of the other clauses in this semantics are self-explanatory.

It is straight-forward to prove a Church-Rosser theorem about the rewrite rules in Figures 2.13 and 2.14. The proof reduces to showing that Id has a one-step Church-Rosser property from which the desired theorem follows by pasting together diamonds as in proofs of the Church-Rosser theorem for lambda-calculus [6]. More precisely, we have the following development.

Definition 23 *$\langle D_1, e_1, \rho_{F1}, \rho_1, FL_1\rangle$ and $\langle D_2, e_2, \rho_{F2}, \rho_2, FL_2\rangle$ are* alpha-*equivalent if $\exists x_1 \ldots x_n \in FL_1,; y_1 \ldots y_n \in FL_2$ such that $FL_1 - \{x_1 \ldots x_n\} = FL_2 - \{y_1 \ldots y_n\}$, and replacing $x_1 \ldots x_n$ by $y_1 \ldots y_n$ in $D_1, e_1, \rho_{F1}, \rho_1$ gives $D_2, e_2, \rho_{F2}, \rho_2$ respectively.*

We assume the existence of a $\xrightarrow{\alpha}$ [6] rule. The following lemma says essentially that Id has a one-step Church-Rosser property. It can be viewed as saying that two enabled reductions do not interfere with each other.

Lemma 14 *Let* $\langle D_0, e_0, \rho_{F0}, \rho_0, FL_0\rangle \rightarrow conf_1$, $\langle D_0, e_0, \rho_{F0}, \rho_0, FL_0\rangle \rightarrow conf_2$. *Then, one of the following holds:*

1. *If* $conf_1$ = error , $conf_2 \rightarrow$ error
2. *If* $conf_2$ = error , $conf_1 \rightarrow$ error
3. *Let* $conf_1 = \langle D_1, e_1, \rho_{F1}, \rho_1, FL_1\rangle$, $conf_2 = \langle D_2, e_2, \rho_{F2}, \rho_2, FL_2\rangle$, *and* $(FL_0 - FL_1)\bigcap(FL_0 - FL_2) = \emptyset$. *Then one of the following holds:*
   (a) $conf_1 \xrightarrow{\alpha} conf_2$
   (b) $(\exists conf_3)\ [conf_1 \rightarrow conf_3 \wedge conf_2 \rightarrow conf_3]$

Proof: The proof follows immediately from a case-by-case analysis of the rules in Figures 2.13 and 2.14, and is omitted. ∎

2.2.4 CHAM Operational Semantics of Id

In this section, we give an operational semantics for Id in the formalism of the Chemical Abstract Machine [8].

As in the discussion of the first order fragment, we assume that a program is a set of definitions. Thus, we assume that the final result of a program is deposited in a special first-order variable, say res. The relationship between the Structured operational semantics of the previous section and the approach in this section is roughly as follows. A configuration[3] $\langle D, , \rho_F, \rho, FL\rangle$ is transformed into a set of equations induced naturally by $D \cup \rho \cup \rho_F$.

As before, define the syntactic categories $Term, Defs$:

$$\begin{aligned} e \in Term \ &::= \ Var|\ exp_1(exp_2)|\ op(e_1, e_2)|\ Array(t)|\ cond(e_1, e_2, e_3)|\ c| \\ &\qquad E; e|\ exp_1(exp_2)|\ (\lambda x.exp) \\ \Theta \in Defs \ &::= \ \Theta_1, \Theta_2|\ var = e \end{aligned}$$

[3]There is no expression component of the configuration, since we are assuming that a program is set of definitions

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

if $\mathcal{V}(x)$ is defined

Op: 1. $< D, e_1 \; op \; e_2, \rho_F, \rho, FL > \rightarrow < D^*, x_1 \; op \; x_2, \rho_F, \rho^*, FL^* >$

where $\{x_1, x_2\} \subseteq FL$, $FL^* = FL - \{x_1, x_2\}$, $\rho^* = \rho \cup \{\{x_1\}, \{x_2\}\}$

$D^* = D \cup \{x_1 = e_1, x_2 = e_2\}$

2. $< D, m \; op \; n, \rho_F, \rho, FL > \rightarrow < D, r, \rho_F, \rho, FL >$

if $r = m \; op \; n$

Con: 1. $< D, cond(e, e_2, e_3), \rho_F, \rho, FL > \rightarrow < D^*, cond(x, e_2, e_3), \rho_F, \rho^*, FL^* >$

if $x \in FL$, $FL^* = FL - \{x\}$, $\rho^* = \rho \cup \{\{x\}\}, D^* = D \cup \{x = e\}$

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

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

Arr: 1. $< D, array(e), \rho_F, \rho, FL > \rightarrow < D \cup \{x = e\}, array(x), \rho_F, \rho^*, FL^* >$

where $x \in FL$, $FL^* = FL - \{x\}$, $\rho^* = \rho \cup \{\{x\}\}$

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

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

3. $< D, e_1[e_2], \rho_F, \rho, FL > \rightarrow < D^*, x_1[x_2], \rho_F, \rho^*, FL^* >$

where $\{x_1, x_2\} \subseteq FL$, $FL^* = FL - \{x_1, x_2\}$, $\rho^* = \rho \cup \{\{x_1\}, \{x_2\}\}$

$D^* = D \cup \{x_1 = e_1, x_2 = e_2\}$

4. $< D, [L1, ..., Ln][i], \rho_F, \rho, FL > \rightarrow < D, Li, \rho_F, \rho, FL >$

where $1 \leq$ i $\leq$ n.

Fun: 1. $< D, e_1(e_2), \rho_F, \rho, FL > \rightarrow < D^*, x_1(x_2), \rho_F, \rho, FL^* >$

where $\{x_1, x_2\} \subseteq FL$, $FL^* = FL - \{x_1, x_2\}$,

$D^* = D \cup \{x_1 = e_1, x_2 = e_2\}$

2. $< D, (\lambda x.e_1)e_2, \rho_F, \rho, FL > \rightarrow < D, y = e_2 \text{ in } e_1^*, \rho_F, \rho^*, FL^* >$

where $y \in FL$, $FL^* = FL - \{y\}$, $e_1^* = e_1[y/x]$, $\rho^* = \rho \cup \{\{y\}\}$

3. $< D, x = e_2 \text{ in } e_1, \rho_F, \rho, FL > \rightarrow < D^*, \mathrm{e1}, \rho_F, \rho, FL >$

$D^* = D \cup \{x = e_2\}$

Figure 2.13: Structured Operational Semantics of Id: Expressions

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

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

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

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

   if x, y first order, $\mathcal{U}(\rho, \{x, y\})$ inconsistent.

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

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

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

   if x first order, $\mathcal{U}(\rho, \{x, c\})$ inconsistent.

4. $< D \cup \{x = [L1, ..., Ln]\}, e, \rho_F, \rho, FL > \rightarrow < D, e, \rho_F, \rho^*, FL >$

   if $\rho^* = \mathcal{U}(\rho, \{x, [L1, ..., Ln]\})$ is consistent

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

5. $< D \cup \{F = \lambda x.e_1\}, e, \rho_F, \rho, FL > \rightarrow < D, e, \rho_F \cup \{F = \lambda x.e_1\}, \rho, FL >$

Figure 2.14: Structured Operational Semantics of Id: Definitions

As in the structured operational semantics, we make a fairness assumption. We assume that there is a notion of fair-scheduling so that no enabled transition is postponed indefinitely.

The structural and unification rules are as in the first order fragment, in Figures 2.7 and 2.5, respectively. The synchronization and process creation rules are described in Figure 2.15. These rules are quite similar to the rules for the first order fragment: the only new rules are the rules for function application.

2.2.5 Denotational Semantics

This section describes the abstract semantics for Id and is organized as follows. We use the example presented in an earlier section to motivate the need for new constructions to model the full language. We use the example to give an informal overview of our approach. Next, we give a formal account of the construction of various domains needed for the semantics and describe the abstract semantics. This section shows show how the higher-order case fits into the picture of constraints and equation solving used in the first order semantics.

1. $e' = (e_1 \; op \; e_2) \rightarrow e' = (x_1 \; op \; x_2), x_1 = e_1, x_2 = e_2$

2. $e' = Array(e) \rightarrow e' = Array(x), x = e$

3. $e' = e_1(e_2) \rightarrow e' = x_1(x_2), x_1 = e_1, x_2 = e_2$

4. $e' = cond(e_1, e_2, e_3) \rightarrow e' = cond(x, e_2, e_3), x = e_1$

5. $e = (\Theta; e') \rightarrow \Theta, e = e'$

6. $y = e, x = v \rightarrow y = e[v/x], x = v$: where v is a base value or abstraction.

7. $e' = cond(true, e_2, e_3) \rightarrow e' = e_2$

8. $e' = cond(false, e_2, e_3) \rightarrow e' = e_3$

9. $e' = (\lambda x.e)e_2, \rightarrow e' = e[x_{new}/x], x_{new} = e_2$

Figure 2.15: CHAM semantics of Id: Process Spawn and Synchronization Rules

Informal Introduction

Consider the following version of the example discussed earlier in Section 2.2.1:

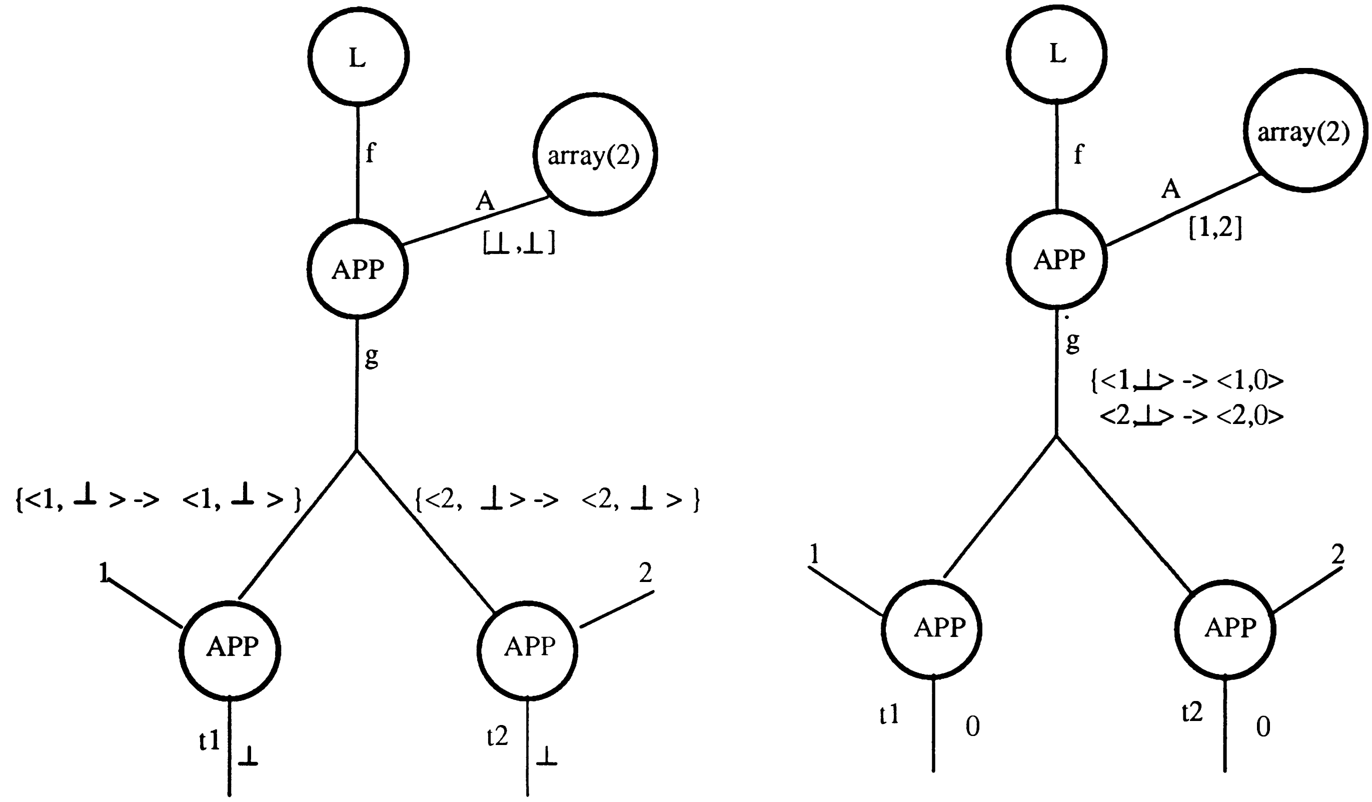
```
def f X i = {X[i] = i in 0}


 {A = array(2);
  g = f A;              ----(5)
  t1 = g 1;             ----(6)
  t2 = g 2;             ----(7)
  in A}
```

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

In a pure functional language, higher-order functions are modeled by currying first-order functions. It is worth understanding why currying is inadequate for modeling the higher-order part of Id. Consider the function $F = \lambda\langle x, y\rangle.e[x, y]$ which represents a function that accepts as input a pair, say of type $D_1 \times D_2$, and returns an element of type D_3. Currying this function gives a function of type $D_1 \to D_2 \to D_3$. If v is of type D_1, the function $G = ((CURRY F) \quad v)$ is of type $D_2 \to D_3$. This type does not model the behavior of functions in the presence of logic variables since it does not reflect the fact that v can get updated when the function G is is applied, as in the example discussed above. In a pure functional language, the value of v does not depend on what happens to G and the function G is determined entirely by F and v. This is not the case once logic variables are introduced; in our example, the value attained by array A depends on the arguments that g has been applied to.

To capture this behavior, we extend the constraint point of view developed for the first-order semantics to functions. In the higher-order semantics, functions like `f` and g are given meanings as *graphs*, and lambda abstractions are given meanings as closure operators on these graphs: for example the graph of g will be a set of elements of the form $\langle u, v\rangle \to \langle u', v'\rangle$. The intuition is that each such pair represents a piece of information about g : given an approximation u to the argument and v to the result, g refines the argument to u' and the result to v'. Function graphs get refined



INTERMEDIATE STAGE

FINAL ANSWERS

Figure 2.16: Dataflow graph for example

through application and this refinement occurs in two ways — the domain of the graph can increase or a particular element $\langle u,v\rangle \rightarrow \langle u',v'\rangle$ gets refined to $\langle u,v\rangle \rightarrow \langle u'',v''\rangle$, where $\langle u',v'\rangle \sqsubseteq \langle u'',v''\rangle$. To understand this better, consider Figure 2.16 which shows a dataflow-like representation of the example. Application nodes are made explicit as *App*, and the term λ `X.` λ `i.  X[i]=i  in  0` is denoted by `L`. Initially, the graphs of `f` and `g` are { } and all other variables have the value $\perp$. The two applications of `g` examine their arguments and results and add the elements $\langle 1,\perp\rangle \rightarrow \langle 1,\perp\rangle$ and $\langle 2,\perp\rangle \rightarrow \langle 2,\perp\rangle$ to graph of g. Also, the node `array(2)` makes the graph on its edge $[\perp,\perp]$: the array of two elements, both of which are undefined. These values are shown in the diagram on the left in Figure 2.16.

The application node corresponding to `g =F A` collects the information about the graph of g and $[\perp, \perp]$ and passes it up to the node labelled `L`. Note that the use of graphs allows us to keep track of the arguments that the functions has been applied to. The graph passed to `F` is

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

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

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

This graph is passed down to the application of `f`.

This application node in turn passes down a refined version of the graph of g, namely $\{\langle 1,\perp\rangle \to \langle 1,0\rangle, \langle 2,\perp\rangle \to \langle 2,0\rangle\}$. Furthermore, it refines the value on the edge connected to the node `array(2)` to $[1,2]$. The new value of the graph of g is used to update values at the application sites of g. For example, the application node corresponding to the statement `t2 =g(2)` can now update `t2` to `0`. The graphs at this stage are shown in the diagram on the right in Figure 2.16. Repeating these steps again does not alter any values. Note that the final result yielded agrees with the answer that the operational semantics predicts.

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

The Semantic Domain

This section is a formal definition of the domain of graphs introduced informally in Section 2.2.5. In the discussion below, D is a complete algebraic lattice. We denote the finite elements of a domain D by $B(D)$; given a set of ordered pairs S, we denote by $Dom(S)$ the domain of S; more precisely: $Dom(S) = \{x \mid (\exists)\ \langle x,y\rangle \in S\}$

Definition 24 *Let D be a domain. Then, the domain of graphs of closure operators on D, denoted $\mathcal{CG}(D)$, is defined as follows. The elements are subsets S of elements of the form $\langle x,x'\rangle$, where $x,x' \in B(D)$, satisfying the following requirements:*

1. *Function:* $\{\langle x, x'\rangle, \langle x, x''\rangle\} \subseteq S \Rightarrow \langle x, x' \sqcup x''\rangle \in S$.

2. *Monotonicity:* $[\{\langle x, x'\rangle, \langle y, y'\rangle\} \subseteq S \wedge y \sqsubseteq x \sqsubseteq y'] \Rightarrow \langle x, y'\rangle \in S$.

3. *Extensivity:* $\langle x, x'\rangle \in S \Rightarrow [x \sqsubseteq x' \wedge x' \in Dom(S)]$

4. *Idempotence:* $\{\langle x, x'\rangle, \langle x', x''\rangle\} \subseteq S \Rightarrow \langle x, x''\rangle \in S$

5. *$Dom(S)$ is downward closed.*

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

The first requirement ensures that we can view graphs as encoding functions — given an element in the domain of the graph, the corresponding output is the most defined element associated with that element by the graph. Taking advantage of this, we will sometimes write $x_1 \rightarrow x_2$ when the pair $\langle x_1, x_2\rangle$ occurs in a graph and x_2 is the most defined element associated with x_1. The fourth requirement, together with the first, ensures idempotence. These requirements are reasonable since we are dealing with graphs of closure operators.

The final requirement is that when an element appears in the domain of the graph, all elements less than it also appear in the domain; this is justified from the operational intuition that if we apply a function to an argument, we have in effect applied it to all values less defined than the argument.

Given a subset S of pairs of elements from $B(D)$, let $\overline{S}$ denote the closure of S under the requirements placed on function graphs; that is, it is smallest element of $\mathcal{CG}(D)$ containing S. If S is a singleton set $\{x\}$, we will sometimes write $\overline{x}$ instead of $\overline{\{x\}}$.

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

Lemma 15 *Let D be a complete algebraic lattice. Then, $(\mathcal{CG}(D), \subseteq)$, is a complete algebraic lattice, with*

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

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

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

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

Function spaces: $\mathcal{D}\sigma_1 \to \sigma_2 = \mathcal{CG}(\mathcal{D}_{\sigma_1} \times \mathcal{D}_{\sigma_2})$ Thus, elements of $\mathcal{D}\sigma_1 \to \sigma_2$ are sets of elements of the form $\langle x, y\rangle \to \langle x', y'\rangle$, where $x, x' \in B(\mathcal{D}_{\sigma_1})$, $y, y' \in B(\mathcal{D}_{\sigma_2})$, satisfying the requirements of Definition 24.

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

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

Lemma 16 *For all types $\sigma_1 \to \sigma_2$, $(\mathcal{D}\sigma_1 \to \sigma_2, \subseteq)$, is a complete algebraic lattice, with*

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

Next, we define some auxiliary functions on domains of graphs that are useful in the presentation of the semantics. The first function is an extension of the operator that performs closure under the requirements on function graphs. The second function, *App*, defines the notion of application for graphs — note that applying a graph can change the graph itself, which is exactly the behavior we required in our informal discussion in Section 2.2.5.

1. Let $u \in B(\mathcal{D}_\sigma), v \in B(\mathcal{D}_\tau)$. Let $u' \in \mathcal{D}_\sigma, v' \in \mathcal{D}_\tau$ be such that $u \sqsubseteq u', v \sqsubseteq v'$. Then, denote by $\langle u, v\rangle \hookrightarrow \langle u', v'\rangle$, the element of $\mathcal{D}\sigma \to \tau$ defined as follows:

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

2. The following definition describes the notion of the application of a graph to an argument. Let $s \in D_{\sigma_1 \rightarrow \sigma_2}$, $t \in D_{\sigma_1}$, $u \in D_{\sigma_2}$. Then,

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

The Semantic Clauses

Figure 2.17 describes the denotations of definitions. The environment in which all identifiers are mapped to $\top$ is called $env_\top$. A definition incorporates all the constraints in its right hand side expression and also constrains the identifier on the left hand side. Composition of definitions is treated as a simultaneous fixpoint of closure operators as discussed in the semantics of the first order fragment. As before, the notation **lcs** in front of a set of simultaneous equations involving closure operators stands for the *least common solution* of that set of equations.

Figures 2.18 and 2.19 describe the denotations of all expressions except lambda abstraction. In the meaning of constants, the function K maps syntactic constants to their abstract equivalents. In the rule for conditionals, note that e_2 and e_3 play no role if e_1 is undefined. Function application is the only tricky clause since application may cause the meaning of the function to change. A simple way to make sense of this rule is to write application as $Apply(e_1, e_2)$, using a prefix *Apply* operator. *App*, the closure operator that is the meaning of *Apply*, was defined in Section 2.2.5 and enforces constraints between e_1, e_2 and the output, refining the value of e_1 if necessary.

Figure 2.20 describes the denotation of lambda abstraction. Recall from the operational semantics that the body of a lambda expression is accessed only when it is applied to an argument. This is mirrored in the denotational semantics in the check for non-empty argument graph. Thus the constraints imposed by the body of the lambda expression are taken into account only when the argument graph is non-empty. Otherwise, we first compute the updated environment using the function *UpdateEnv* which essentially evaluates the body of the lambda expression in each environment obtained by binding the formal parameter to an actual parameter obtained from a, the approximation to the graph. The new environment is used to compute the new value of the

$$
\begin{aligned}
\mathcal{C}[\![x = e]\!]\ env &= \text{lcs}\begin{cases} env \sqsubseteq env' \\ \langle env', b' \rangle = \mathcal{E}[\![e]\!]\langle env', b\rangle \\ env'[x] = b \end{cases} \\
&\quad \text{in } env' \\
\mathcal{C}[\![def_1\ ;\ def_2]\!]\ env &= \text{lcs}\begin{cases} env \sqsubseteq env' \\ env' = \mathcal{C}[\![def_1]\!]\ env' \\ env' = \mathcal{C}[\![def_2]\!]\ env' \end{cases} \\
&\quad \text{in } env'
\end{aligned}
$$

Figure 2.17: Denotational Semantics of Id: Definitions

graph. Notice that there is no special case for recursion. The case of recursion is handled implicitly by the definition of the denotation of equations. This is analogous to the handling of feedback loops by a fixpoint iteration in static determinate Kahn dataflow. The fixpoint iteration in this case is performed in the computation of the least common solution.

Properties of the semantics

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

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

For a general closure operator f, we can write this condition as follows: $\langle env, a\rangle \in Fix(f)\ \wedge\ \langle env, b\rangle \in Fix(f) \Rightarrow \langle env, a \bigsqcup b\rangle \in Fix(f)$. We call this condition *additivity.* It is easy to check that the parallel composition of additive closure operators is additive. Furthermore, the additive closure operators are closed under limits of directed sets. It is easy to see that all the closure operators that are denotations of expressions satisfy this condition.

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

Figure 2.18: Denotational semantics of Id: Expressions

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

Figure 2.19: Denotational Semantics of Id: Array expressions

2.2.6 Relating the Semantic Definitions

In this section we prove that the denotational semantics and the operational semantics coincide. The main result of this section is the adequacy: two programs are operationally identical only if they have the same denotation. This section is organized as follows: first, we present an outline of the proof; next, we present the proof in full detail.

Proof Outline

This section presents the informal sketch of the proof that the denotational semantics is correct for reasoning about equality in the operational semantics.

One step reduction preserves meaning

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

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

$$
\begin{aligned}
UpdateGraph^e(env')(\overline{(\langle u, v\rangle \rightarrow \langle u', v'\rangle)}) &= \text{lcs}\begin{cases} env'[x \mapsto \overline{u'}] \sqsubseteq env^\star \\ \overline{v'} \sqsubseteq b_r \\ \langle env^\star, b_r\rangle = \mathcal{E}[\![e]\!]\langle env^\star, b_r\rangle \end{cases} \\
&\quad\ \text{in } (\langle u, v\rangle \hookrightarrow \langle env^\star[x], b_r\rangle) \\
UpdateGraph^e(env')(\overline{\{g_1 \ldots g_n\}}) &= \bigsqcup_i\{UpdateGraph^e(env')(\overline{g_i})\} \\
UpdateGraph^e(env')(S) &= \bigsqcup\{UpdateGraph^e(S_f) \mid S_f \sqsubseteq_f S\}
\end{aligned}
$$

Figure 2.20: Denotational Semantics of Id: Lambda terms

preserves meaning.

In order to show that one-step reduction preserves meaning we need to associate meanings with the basic entities used in the operational semantics, i.e. with configurations. In the following, the semantic function $\mathcal{M}$ assigns to configurations a closure operator over the domain $ENV \times_{\top} V$. We use the semantic functions $\mathcal{E}$ and $\mathcal{C}$ defined previously and the same notational conventions.

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

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

The Adequacy Theorem

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

We use the relationship $\preceq$ between first order syntactic expressions, e, and closure operators, f, on $ENV \times_{\top} V$, defined earlier. Recall that, $\mathcal{E}[\![e]\!] \preceq e$ intuitively means that given any finite approximant to the result predicted by $\mathcal{E}[\![e]\!]$, there is a finite sequence of reductions evaluating e in a suitable syntactic environment, that produces a more refined value. In particular, if the result predicted by $\mathcal{E}[\![e]\!]$ is $\top$, evaluating e in a suitable syntactic environment results in *error*. In this section, we extend the first order result to the full higher order language. This is done using the idea of logical relations used in the proofs of adequacy in usual functional languages [53]. Thus, we define a notion of computability of terms by induction on types.

Definition 25 *(Definition of the computability predicate for closed terms)*

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

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

Theorem 5 *All terms are computable.*

Full Abstraction

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

One-step reduction preserves meaning

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

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

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

In order to show that one-step reduction preserves meaning we need to associate meanings with the basic entities used in the operational semantics, i.e. with configurations. In the following the semantic function $\mathcal{M}$ assigns to configurations a closure operator of suitable type. We use the semantic functions $\mathcal{E}, \mathcal{F}$ and $\mathcal{C}$ used in the denotational semantics and the same notational conventions.

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

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

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

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

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

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

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

Theorem 6 *Suppose that the following rewrite is possible:*

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

Let env satisfy the following conditions:

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

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

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

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

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

Then, we have the following:

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

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

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

These proofs are described in the following two subsections. ∎.

"β"-reduction preserves meaning

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

$$< D, (\lambda x.e_1)e_2, \rho_F, \rho, FL > \rightarrow < D, y = e_2 \text{ in } e_1^*, \rho_F, \rho^*, FL^* >$$

where $y \in FL$, $FL^* = FL - \{y\}$

where $e_1^* = e_1[y/x]$, $\rho^* = \rho \cup \{\{y\}\}$

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

Lemma 17 *Let $env[x] = \perp$, and x not free in e_2. Then,*

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

where env' differs from env only at x.

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

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

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

$$\langle env, c\rangle\rangle = \mathcal{E}[\![e_2]\!]\langle env, c\rangle \quad 1$$

$$\langle env, b\rangle\rangle = \mathcal{E}[\![\lambda x.e_1]\!]\langle env, b\rangle \quad 2$$

$$(b, c, a) = App(b, c, a) \quad 3$$

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

$$\langle env', c'\rangle\rangle = \mathcal{E}[\![e_2]\!]\langle env', c'\rangle \quad 4$$

$$\langle env', a'\rangle\rangle = \mathcal{E}[\![e_1]\!]\langle env', a'\rangle \quad 5$$

$$\langle env', c'\rangle\rangle = \mathcal{E}[\![x]\!]\langle env', a'\rangle \quad 6$$

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

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

$$[\langle c'_1, a'_1 \rangle \rightarrow \langle c'_2, a'_2 \rangle \sqsubseteq b' \wedge \langle c'_1, a'_1 \rangle \sqsubseteq \langle c', a' \rangle] \Rightarrow \langle c'_2, a'_2 \rangle \sqsubseteq \langle c', a' \rangle$$

The definition of $UpdateGraph^{e_1}(env')(\langle c'_1, a'_1 \rangle \rightarrow \langle c'_1, a'_1 \rangle)$ is,

$$UpdateGraph^{e_1}(env')(\langle c'_1, a'_1 \rangle \rightarrow \langle c'_1, a'_1 \rangle) \;=\; \begin{array}{l} \text{lcs} \left\{ \begin{array}{l} env'[x \mapsto \overline{c'_1}] \sqsubseteq env^\star \\ \overline{a'_1} \sqsubseteq a'_r \\ \langle env^\star, a'_r \rangle \rangle = \mathcal{E}[\![e]\!]\langle env^\star, a'_r \rangle \\ (\langle c'_1, a'_1 \rangle \hookrightarrow \langle env^\star[x], b \rangle) = b \end{array} \right. \\ \text{in } b \end{array}$$

Note that $a'_1 \sqsubseteq a' = env'[x]$ and $c'_1 \sqsubseteq c'$. Thus, in the above $a'_r \sqsubseteq a', env^\star[x] \sqsubseteq c'$. Hence, $\langle c'_2, a'_2 \rangle \sqsubseteq \langle c', a' \rangle$, and result follows.

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

$$\langle env, b \rangle \rangle = \mathcal{E}[\![\lambda x.e_1]\!]\langle env, b \rangle \quad 2$$

$$(b, c, a) = App(b, c, a) \quad 3$$

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

Replacing names of function symbols by definitions

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

$< D, C[x], \rho_F, \rho, FL > \rightarrow < D, C[\mathcal{V}(x)/x], \rho_F, \rho, FL >$

if $\mathcal{V}(x)$ is defined

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

Lemma 18 *Let L be an expression of from $\lambda x.e$. Let env, a_t, a_{arg}, a_r satisfy the following:*

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

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

1. *Forall expressions e' in which F occurs only in subexpressions of the form $F(e'')$, $\mathcal{E}[\![e']\!]\langle env, b_t\rangle = \langle env, b_t\rangle \Rightarrow \mathcal{E}[\![e']\!]\langle env_{new}, b_t\rangle = \langle env_{new}, b_t\rangle$*
2. *$\mathcal{E}[\![L = F]\!]\ env_{new} = env_{new}$.*

Proof: The proof involves reasoning about the semantic definitions.

1. The first step of the proof is to show that:

$$\overline{\langle u, v\rangle \to \langle u', v'\rangle} \sqsubseteq a_t \wedge \langle u, v\rangle \in DOM(env[F]] \Rightarrow \overline{\langle u, v\rangle \to \langle u', v'\rangle} \sqsubseteq env[F]$$

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

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

2. From hypothesis of the theorem, $\langle env, env[F]\rangle \in Fix(\mathcal{E}[\![L]\!])$. Let $\langle env, a_t\rangle \in Fix(\mathcal{E}[\![L]\!])$. Since L is of the from $\lambda x.e$, and L and F are of the same type, all occurrences of F in L occur in an applicative context. From the first part of the

theorem, we deduce that $\langle env_{new}, env[F]\rangle \in Fix(\mathcal{E}[\![L]\!])$. Similarly, $\langle env, a_t\rangle \in Fix(\mathcal{E}[\![L]\!]) \Rightarrow \langle env_{new}, a_t\rangle \in Fix(\mathcal{E}[\![L]\!])$. Using additivity, $\langle env_{new}, env[F] \bigsqcup a_t, \rangle \in Fix(\mathcal{E}[\![L]\!])$. Hence, $\mathcal{E}[\![L = F]\!]\ env_{new} = env_{new}$.

Hence the result. ∎

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

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

Definition 26 *env is* valid *for e, if forall function symbols F such that $F = L$ is a subexpression of e, $\mathcal{E}[\![F = L]\!]env[F \mapsto [DOM(env[F]) \rightarrow DOM(env[F])]] = env$.*

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

Lemma 19 *Let env be* valid *for e and* $\mathcal{E}[\![e]\!]\langle env, a\rangle = \langle env', b\rangle$*. Then, env' is valid for e.*

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

Note that the only environments that we are interested in are those got by executing a program in an uninitialized environment. Thus, lemma 19 allows us to deduce that all the environments that we are interested are *valid* for the relevant expressions.

Lemma 20 *Let L be an expression of form* $\lambda x.e$*. Let* env, a_t *satisfy the following conditions:*

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

Then, $\langle a_{arg}, b\rangle \rightarrow \langle a_{arg}, b\rangle) \sqsubseteq a_t \Rightarrow App(env[F] \sqcup a_t, a_{arg}, b) = App(a_t, a_{arg}, b)$.

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

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

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

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

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

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

Thus, the iterates can be rewritten as

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

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

Adequacy

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

The proof of adequacy uses the idea of logical relations used in the proofs of adequacy for the simply typed lambda calculus [53] to lift the adequacy proof for first order terms to terms of higher types. The rest of this section is organized as follows. In the first subsection, we define inclusive predicates to relate syntactic and semantic entities. In the next subsection, we prove the adequacy theorem.

Relating syntactic and semantic values

In this section, we extend the relation $\preceq$ to work for higher order objects. This is done following the idea of logical relations used in the adequacy proofs for typed functional languages [53]. The idea is to "lift" the first order definition to expressions of higher type. The following definition proceeds by induction on formation of types. The base case of the definition uses the inclusive predicate defined in the previous subsection. Define a predicate $Comp$ by induction on types, on closed terms as follows:

Definition 27 *(Definition of the computability predicate)*

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

The definition is extended to open terms through the notion of valid substitutions.

Definition 28 *(Valid Substitutions)*
Let $\{x_1^{\sigma_1} \dots x_n^{\sigma_n}\}$ be a set of variables, Then, a set of equations E is a valid substitution for $\{x_1^{\sigma_1} \dots x_n^{\sigma_n}\}$ if

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

$$Comp(e_j) \vee [Comp(y_j) \Rightarrow Comp(e_j)]$$

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

Definition 29 *Let e be an open term with $FV(e) = \{x_1 \dots x_n\}$. Then, $Comp(e)$ if forall valid substitutions E, $Comp(E$ in $e)$.*

Adequacy Proof

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

The cases of structural induction for the first order combinators has been proved earlier. This lemma is the base case for the adequacy proof for the higher order language.

Lemma 21 *(First order properties)*

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

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

Lemma 22 *Let*

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

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

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

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

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

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

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

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

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

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

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

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

Lemma 23 *Variables of non-base type are computable.*

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

1. From definition of valid substitutions, it follows that, forall n, the terms $F_0 = \lambda x.x; F_1 = L[F_0]; \ldots F_n = L[F_{n-1}]$ in F_n are computable

2. $\langle D[F_n], e[F_n], \rho, \rho_F^*, FL\rangle \xrightarrow{*} \langle D'[F_0], e[F_0], \rho, \rho_F^{**}, FL\rangle$ implies that

$$\langle D[F], e[F], \rho, \rho_F, FL\rangle \xrightarrow{*} \langle D'[F], e[F], \rho, \rho'_F, FL\rangle$$

where the starred ρ_F's differ from the the corresponding unstarred ρ_F's only in the bindings of $F, F_0 \ldots F_n$: the proof is a straightforward inductive argument on the length of the reduction sequences.

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

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

Result follows from the two observations above. ∎

Theorem 7 *All terms are computable.*

Proof: Proof is by structural induction. From the lemmas above, all variables are computable. From lemma 21 all first order constructors form computable expressions from computable expressions. The cases of structural induction for E **in** e and $\lambda x.e$ and $e(e')$ follow straightforwardly from definitions. ∎

2.3 Conclusions

This chapter describes an investigation into an abstract semantics for Id, a functional language with logic variables. The operational semantics was presented in a Plotkin-style structured operational semantics and in the style of the Chemical Abstract Machine. The operational semantics incorporated an explicit treatment of aliasing introduced by unification, and the interleaving of computations in different parts of the program. The abstract semantics showed that Id programs could be given a simple denotational treatment couched entirely in terms of equations and equation solving. The denotational semantics was proved to be fully abstract with respect to the operational semantics for the first order fragment. Furthermore, the denotational semantics was shown to be adequate with respect to the operational semantics for the full language.

The semantic techniques developed here can be applied to any language in which objects are created through constraint intersection. Recent research [57] describes a general notion of a constraint system incorporating conjunction and hiding. This paper also shows that closure operators form the right models for deterministic concurrent constraint programming.

The full abstraction result for the first order language Cid is interesting in the context of research into full abstraction for functional languages [53,42,16]. Full abstraction for functional languages was first studied by Plotkin in his seminal paper on LCF where full abstraction was obtained by adding parallel-or construct to the language [53]. Intuitively, the parallel-or operator allowed the syntactic definition of the semantic least upper bound function. The semantics of the first order language is fully abstract without the addition of new syntactic constructs because the least upper bound function is introduced implicitly into the language through composition of definitions. This can be viewed as semantic evidence of the parallelism in the operational model of the language.

The semantic description of this chapter extends the equation-solving paradigm that underlies Kahn semantics for dataflow networks [32] to a more expressive setting in which processes manipulate shared memory locations. Kahn's dataflow networks communicate by sending messages on channels, and message transmission is monotonic in the sense that a message cannot be recalled once it has been sent. Our framework extends this model with monotonic shared memory. This extension is non-trivial because it allows the communication abilities of processes to change dynamically, unlike the Kahn model of dataflow in which the channel structure of networks is fixed and cannot be altered during runtime.

The techniques of this chapter have been extended to languages with notions of search, akin to the notion of search in logic programming [30]. The extension of this work to the indeterminate setting is still in a state of flux [17,57]. Chapter 4 of this thesis presents an alternate approach to such languages.

Chapter 3

A Domain-theoretic Model for a Higher-order Process Calculus

3.1 Introduction

In this chapter[1] we study a *higher-order* process calculus, a restriction of one due to Boudol [10,8] and develop an abstract model for it. By abstract we mean that the model is constructed domain-theoretically and reflects a certain conceptual viewpoint about observability. It is not constructed from the syntax of the calculus or from computation sequences. We describe a new powerdomain construction that can be given additional algebraic structure that allows one to model concurrent composition, in the same sense that Plotkin's powerdomain can have a continuous binary operation defined on it to model choice. We show that the model constructed this way is adequate with respect to the operational semantics. The model that we develop and our analysis of it is closely related to the work of Abramsky and Ong [4,50] on the lazy lambda calculus.

Our study of the restricted version of Boudol's calculus, henceforth called the γ-calculus, is based on viewing the communication ability of processes as the fundamental observable. A process that is diverging has no communication ability, a process that can accept a single input and then diverges has more communication ability. This is a natural extension of the idea of making convergence the basic observable in the λ-calculus. The connection with the lazy λ-calculus comes about by observing that

[1]The material in this chapter is drawn from joint work with Prakash Panangaden [26].

the presence of an outer λ-abstraction signifies that a term has communication ability. Clearly, we should distinguish $\lambda x.\Omega$ from Ω, where Ω represents any divergent term such as $(\lambda x.xx)(\lambda x.xx)$, since they have different communication abilities. Thus we need our model to resemble the models of the lazy λ-calculus [4] rather than the models discussed by Scott and Wadsworth [68].

We model concurrency in the γ-calculus as indeterminate interleaving. Thus, we need to deal with the fact that a term may or may not converge. We need two predicates to capture the convergence properties of term; these are "may converge", written $\Downarrow^{may}$, and "must converge", written $\Downarrow^{must}$. The operational preorder is defined in terms of these predicates.

The main contribution of this chapter is the new powerdomain construction that we describe. Roughly speaking, it allows us to model processes as sets of functions and allows us to capture the notions of observability described above. Intuitively, the key difference between our powerdomain and the Plotkin powerdomain [52] is that our construction is defined to work on function spaces. One cannot, of course, expect the last remark to be taken literally since when one is handed a domain, even one that is a function space, it may not be presented as a function space. The recursive domain equation that we solve uses a functor that first builds a function space and then carries out certain constructions on the result.

One important point worth stressing early is why we defined a new powerdomain instead of using the Plotkin powerdomain. Both powerdomains would yield an adequate model and, as far as we know, neither yields a fully abstract model. Nevertheless, we feel that our model comes closer to capturing the operational properties of the calculus. When one attempts to use the Plotkin powerdomain to model the γ calculus, one finds that certain operational laws are violated in the model. Concretely, we exhibit terms that are deemed equal by the operational semantics, such that the meanings of these terms are unrelated in a model based on the Plotkin powerdomain, and are the same in our semantics. In fact, we can prove that the model that we construct is "internally fully abstract" [4]. More precisely, we can define an operational preorder on terms that embodies the above notions of observability. It turns out that this preorder meshes well with the partial order in our model in the following sense. Suppose that we define a preorder on domain elements that formally imitates the definition of the operational preorder. Then, we recover exactly the original partial order in the model. This does not happen with the Plotkin powerdomain. Thus, our model, though probably not fully abstract, describes the interplay between choice, lambda abstraction and concurrency in a smooth way. This discussion is made precise in later sections.

This chapter is organized as follows. We introduce the γ-calculus and discuss it informally. Next, we define a sub-calculus, its operational semantics and introduce the operational preorder. Next, we give the powerdomain construction and delineate the algebraic properties of the model. Finally, we describe the adequacy properties of the model.

3.2 The γ Calculus

In this section we quickly review Boudol's γ-calculus and describe an example of a simple concurrent program expressed in it. The key contribution of this calculus is to provide a smooth integration of concurrent communication concepts with functional abstraction. Boudol's original work [10] describes the calculus and shows how the lazy λ-calculus is embedded in it.

Let C be a set of channel names. Terms are generated by the grammar:

$$\begin{aligned} Terms \ ::= \ & x \ || \ (\lambda_1 x_1 | \ldots | \lambda_k x_k).p \ || \\ & p \odot \ q \ || \ p|q \ || \ \overline{\lambda} p.1 \ || 1 \end{aligned}$$

where λ, $\lambda_1 \ldots \lambda_k$ are (not necessarily distinct) members of C. The novel constructs here are $(\lambda_1 x_1 | \ldots | \lambda_k x_k)$, $\odot$, $\overline{\lambda}$ and $p|q$. The term 1 represents the terminated process. It will turn out to be the identity for both $\odot$ and $|$. Roughly speaking, $(\lambda_1 x_1 | \ldots | \lambda_k x_k).p$ means that p waits *concurrently* for k unordered values. The combinator$|$ represents concurrency. The intuitive meaning of $p|q$ is that p and q are juxtaposed, *without any communication between them.* The term $\overline{\lambda} p$ represents a process that outputs p on channel λ and terminates. Finally, $p \odot q$ means that p and q communicate on *all* channels. The processes p and q cannot communicate with any other process until one of them terminates. The key points to note are that $|$ represents pure concurrency without any interaction while $\odot$ represents a very tight interaction between processes.

The following transition system presented informally, expresses these intuitive ideas. First we begin by defining a syntactic congruence that expresses the fact that 1 is the terminated process.

Definition 30 *The syntactic relation $\equiv$ is the congruence (with respect to substitution) that is generated by the following equations:*

- $p \odot 1 \;\equiv\; 1 \odot p \;\equiv\; p$
- $p|1 \;\equiv\; 1|p \;\equiv\; p$
- $p|(q|r) \equiv\; (p|q)|r$

Let λ_i be channel names.

- $(M_1|\ldots|(\lambda_1 x_1|\ldots|\lambda_k x_k).M|\ldots|M_n) \odot\; (N_1|\ldots|\overline{\lambda_i}N|\ldots|N_r)$
  $\rightarrow\; (M_1|(\lambda_1 x_1\ldots|\lambda_{i-1}x_{i-1}|\lambda_{i+1}x_{i+1}\ldots|\lambda_k x_k)[x_i \mapsto N]M|M_{k+1}\ldots M_n)$
  $\odot\;\; (N_1|N_2|\ldots N_s|N_{s+1}\ldots N_r)$

- $(N_1\ldots N_s|\overline{\lambda_i}N|N_{s+1}\ldots N_r) \odot (M_1\ldots M_k|(\lambda_1 x_1\ldots\lambda_k x_k).M)|M_{k+1}\ldots|M_n)$
  $\rightarrow(N_1\ldots N_s|N_{s+1}\ldots N_r) \odot$
  $(M_1\ldots M_k|(\lambda_1 x_1\ldots\lambda_{i-1}x_{i-1}|\lambda_{i+1}x_{i+1}\ldots\lambda_k x_k).[x_i \mapsto N]M|M_{k+1}\ldots M_n)$

-
  - $M\rightarrow M' \Rightarrow M|N\rightarrow M'|N$
  - $N\rightarrow N' \Rightarrow M|N\rightarrow M|N'$

-
  - $M\rightarrow M' \Rightarrow M \odot N\rightarrow M' \odot N$
  - $N\rightarrow N' \Rightarrow M \odot N\rightarrow M \odot N'$

In the above $[x \mapsto N]M$ is notation for substitution. The $\odot$ serves as a generalization of application. The communication is effected in the manner now customary in process algebras, one matches a name with its *dual* name. Note how there is no communication between processes that are combined with $|$. Finally there is no construct like $\overline{\lambda}p.M$ where M represents a term. An output term cannot produce a value and go on to do something else.

The following simple term that appers in Boudol's original paper [10], illustrates some of the features of the γ-calculus.

$$A \simeq \lambda y.\alpha x.(\overline{\beta}z|(y \odot \overline{\lambda}y))$$

Now consider $A \odot \overline{\lambda} A$. This term reduces in one step to

$$\alpha x.(\overline{\beta} z|(A \odot \overline{\lambda} A)))$$

This last term has the property that it waits for a signal on α then outputs z on β and reproduces itself. It is a term that repeatedly offers communication to the outside.

3.3 Operational Semantics

In this section we define the restricted calculus, its reduction rules and a notion of observations.

3.3.1 Syntax and reduction rules

From the point of view of difficulty of modeling we have eliminated the possibility of deadlock but we still have indeterminacy and concurrency. We do not allow $\odot$ in its unrestricted form. We force it to look like applications. More precisely, the $\odot$ construct can only be used in the combination $\lambda x.M \odot \overline{\lambda} P$. Thus it cannot be introduced in a case where there is no communication possibility as in $\lambda x.x \odot \lambda x.x$.

The terms are generated by the grammar

$$Terms ::= x \mid\mid \lambda\langle x_1 \ldots x_k\rangle.p \mid\mid (pq) \mid\mid p|q$$

We do not use the $\odot$ symbol explicitly, it is implicitly present in the applications (pq). We usually drop the parenthesis from (pq). We use Λ_0 for the terms that do not have free variables, and call members of this set closed terms.

Definition 31 *The syntactic equality $\equiv$ is the congruence generated by the equation:*

$$p|(q|r) \equiv\ (p|q)|r$$

Define, by mutual recursion:

$Terms_1 ::= x \mid\mid \lambda\langle x_1 \ldots x_k\rangle.p \mid\mid pq$

$Terms_2 ::= p \mid\mid p|q$

where $p,\ q \in Terms_1 \bigcup Terms_2$. Note that $Terms_2 = Terms$. Intuitively, $Terms_1$ are the terms without a $|$ at 'outermost level', and $Terms_2$ are the terms of the form

$t_1|\dots|t_n$, where the t_i are either abstractions or applications. The following definition is intended to capture the "number" of $t_i's$.

Define $len : Terms_2 \rightarrow Int$ as follows:

- $len(p) = 1$, if $p \in Terms_1$
- $len(p|q) = len(p) + len(q)$

It can be checked that this function is well-defined on the terms quotiented by the syntactic equality $\equiv$. The following definition is intended to capture the "position" of t_i in $t_1|\dots t_n$. Define a partial function $index : \omega \times Terms_2 \rightarrow Terms_1$ as follows:

- $index(n,p) =$ undefined if $len(p) \leq n \wedge\ len(p) \neq n$
- $index(1,p) = p$, if $p \in Terms_1$
- $index(n,p|q) = index(n,p)$, if $n \leq len(p)$
- $index(n,p|q) = index(n - len(p), q)$, if $len(p) \leq n \wedge len(p) \neq n$

The reduction rules are as follows. In this presentation, we have introduced notation (as a subscript of $\rightarrow$) to keep track of the redices, explicitly.

- $(\lambda\langle x_1 \dots x_k\rangle.M)N \rightarrow_{\langle 1,i\rangle} \lambda\langle x_1 \dots x_{i-1}, x_{i+1} \dots x_k\rangle.[x_i \mapsto N]M$
  if $1 \leq i \leq k$
- $index(s, (M_1|\dots|\lambda\langle x_1 \dots x_k\rangle.M|\dots|M_n)) = \lambda\langle x_1 \dots x_k\rangle.M \Rightarrow$
  $(M_1|\dots|\lambda\langle x_1 \dots x_k\rangle.M|\dots|M_n)N$
  $\rightarrow_{\langle s,i\rangle}$
  $M_1|\dots|\lambda\langle x_1 \dots x_{i-1}, x_{i+1} \dots x_k\rangle.[x_i \mapsto N]M|\dots|M_n$, if $1 \leq i \leq k$
- $M \rightarrow_\sigma M' \Rightarrow M|N \rightarrow_\sigma M'|N$
- $N \rightarrow_\sigma N' \Rightarrow M|N \rightarrow_{\sigma'} M|N'$ where $\sigma' = \langle first(\sigma) + len(M), second(\sigma)\rangle$
- $M \rightarrow_\sigma M' \Rightarrow MN \rightarrow_{\langle 1,\sigma\rangle} M'N$

Usually, we ignore these subscripts and write the reductions as below.

- $(M_1|\dots|\lambda\langle x_1 \dots x_k\rangle.M|\dots|M_n)N \rightarrow$
  $M_1|\dots|\lambda\langle x_1 \dots x_{i-1}, x_{i+1} \dots x_k\rangle.[x_i \mapsto N]M|\dots|M_n$, if $1 \leq i \leq k$

- $M \rightarrow M' \Rightarrow M|N \rightarrow M'|N$
- $N \rightarrow N' \Rightarrow M|N \rightarrow M|N'$
- $M \rightarrow M' \Rightarrow MN \rightarrow M'N$

3.3.2 Observables and operational equivalence

Recall that the intuitive meaning that was assigned to $\lambda x.M$ was the presence of a communication ability on port λ. We take the point of view that the only observable behavior about a process is the acceptance of values on channels. So, we attempt to set up a theory that "measures" the communication ability of a term. The study of the lazy lambda calculus [4,50] proceeds on very similar lines. There the "definedness" of a term is measured by its outermost abstractions or, in other words, how many arguments it can accept. This is exactly what we do except that we need to confront the indeterminacy in the reduction relation. The study of the lazy λ-calculus motivates the definition of a convergence predicate. Notice that the presence of non-determinism means that for a given term M, we might have both the following situations:

- $M \xrightarrow{*} \lambda\langle x_1 \dots x_k \rangle.M'$
- An infinite reduction sequence
  $M = M_0 \rightarrow M_1 \rightarrow M_2 \dots$

We define predicates $\Downarrow^{may}$ read as "may converge" and $\Downarrow^{must}$ read as "must converge".

$M\Downarrow^{may}$ is intended to indicate that M can accept input on channel λ after a (possibly empty) finite sequence of silent actions. This can be viewed as a kind of "partial correctness assertion" about M.

Definition 32 *Define a set S of terms inductively as follows:*

1. $\lambda\langle x_1 \dots x_k \rangle.M \in S$, $1 \leq k$, $(\forall \lambda\langle x_1 \dots x_k \rangle.M \in \Lambda_0)$
2. $[M \in S] \Rightarrow (\forall N \in \Lambda_0)[M|N \in S]$

$M\Downarrow^{may}$ *if* $(\exists M')\ [M \xrightarrow{*} M' \ \wedge \ M' \in S]$

To model total correctness guarantees on terms, we need to be able to say that a term M "always accepts input on channel λ", as opposed to the M "can accept input on channel λ" assertion that motivated $\Downarrow^{may}$. In the setting of the subcalculus with only one channel, this is equivalent to saying that M has no infinite silent computation.

Definition 33 *$M\Downarrow^{must}$ if there is no infinite reduction sequence $M \rightarrow M_1 \rightarrow \ldots$*

The predicates $\Downarrow^{may}$ and $\Downarrow^{must}$ are the primitive observables in the calculus. Two terms that "behave" similarly in all contexts with respect to this notion of observation, are not to be differentiated. So, we define an operational preorder $\preceq_c$, in the style of definitions of contextual precongruence in the setting of the lambda calculus [6].

Definition 34 *The relation $\preceq_c$ on Λ_0 is defined by $M \preceq_c N \Leftrightarrow (\forall C[.] \in \Lambda_0)$, the following hold:*

1. $C[M]\Downarrow^{may} \Rightarrow C[N]\Downarrow^{may}$
2. $C[M]\Downarrow^{must} \Rightarrow C[N]\Downarrow^{must}$

The idea of $\preceq_c$ is extended to open terms in the usual way. Let M, N be terms such that the free variables of M and N are contained in $\{x_1 \ldots x_n\}$. Then, $M \preceq_c N$ if for all possible substitutions $P_1 \ldots P_n$ of closed terms for $\{x_1 \ldots x_n\}$, we have
$[x_1 \mapsto P_1 \ldots x_n \mapsto P_n]M \preceq_c [x_1 \mapsto P_1 \ldots x_n \mapsto P_n]N$.

We now define a preorder $\preceq$ that relates the communication abilities of terms in purely applicative contexts. It turns out that the preorders $\preceq$ and $\preceq_c$ coincide. This simplifies operational proofs of equality of terms.

Define (on closed terms):

1. $M \preceq_0 N$ if
   - $M\Downarrow^{may} \Rightarrow N\Downarrow^{may}$
   - $M\Downarrow^{must} \Rightarrow N\Downarrow^{must}$
2. $M \preceq_{k+1} N$ if
   - $M \preceq_k N$
   - $(\forall P)\ [MP \preceq_k NP]$

Definition 35 $\preceq = \bigcap \preceq_k,\ k \in \omega$

The relation $\preceq$ has an alternate characterization as the greatest fixed point of a monotone functional.

Lemma 24 *Define a function F on binary relations of closed terms by:*
$M\ F(R)\ N$ if

- $M \preceq_0 N$
- $(\forall P \in \Lambda_0)\ [(MP,\ NP) \in R]$

Then, $\preceq$ is the maximum fixed point of F

Proof: F is monotone on relations ordered by $\subseteq$. From Tarski's fixed point theorem, F has a maximum fixed point. It easily follows from the previous lemma that the closure ordinal of F is in fact ω. ∎

A small example will help to illustrate the nature of the preorder $\preceq$. We abbreviate the term $\lambda\langle x, y\rangle.x$ as *or* and write it in infix form for readability. Let M denote the term $\lambda x.[\Omega\ \mathit{or}\ \lambda y.\Omega]$. Let N denote the term $[\lambda xy.\Omega\ \mathit{or}\ \lambda x.\Omega]$. Then, a simple proof on the inductive definition of $\preceq$ shows that M and N are equivalent under the operational preorder. This identification of choices made "before" and "after" an abstraction will play a key role in the development of our domain theoretic semantics.

3.3.3 Operational extensionality

In this section, we prove that the relations $\preceq$ and $\preceq_c$ coincide. This is called *Operational extensionality.* To prove this, we first prove that the operator $|$ is monotone with respect to $\preceq$. This proof involves a detailed analysis of reductions. The main theorem is then proved by a variant of the proofs of operational extensionality in lambda calculi [7].

Monotonicity of $|$

In this subsection, the monotonicity of $|$ with respect to $\preceq$ is proved. This is a prelude to the major result of this section namely that $\preceq$ is operationally extensional. This section may be skipped on a first reading[2]. The main point is that proving that $|$ is monotone with respect to $\preceq$ requires an analysis of the interleavings of the reductions in each component. The proofs are not hard but they do require a rather careful analysis of reduction.

[2]And in all subsequent readings?

Lemma 25 *1.* $(Q_1|Q_2)\Downarrow^{may} \Leftrightarrow Q_1\Downarrow^{may} \ \vee \ Q_2\Downarrow^{may}$

2. $(Q_1|Q_2)\Downarrow^{must} \Leftrightarrow Q_1\Downarrow^{must} \ \wedge \ Q_2\Downarrow^{must}$

Proof:

1. This follows directly from the definition.

2. Note that the transition system has the rules

   - $M{\rightarrow}M' \Rightarrow M|N{\rightarrow}M'|N$
   - $N{\rightarrow}N' \Rightarrow M|N{\rightarrow}M|N'$

   Also, all transitions of $M|N$ are of the above type. The result follows. ∎

The following lemma involves interleaving the reductions. The basic idea is best illustrated with a simple example. Consider $(Q_1|Q_2)P_1P_2$. It may converge if Q_2P_2 may converge; in order for this to happen, however, it has to be the case that Q_1 may converge in order for it to be possible for Q_1 to accept the argument P_1. Thus one has to describe the effects of partitioning the arguments to a parallel composition in the following fashion.

Lemma 26 *Let* $0 \leq n$. *Then* $(Q_1|Q_2)P_1 \ldots P_n\Downarrow^{may} \ \Leftrightarrow$
$(\exists\langle i_1 \ldots i_k\rangle, \langle j_1 \ldots j_l\rangle$ *such that*

- $\langle i_1 \ldots i_k\rangle$ *and* $\langle j_1 \ldots j_l\rangle$ *are (possibly empty) strictly increasing sequences of integers from* $\{1 \ldots n\}$
- $k + l = n$
- *At least one of the following hold:*

  1. $(Q_1P_{i_1} \ldots P_{i_k})\Downarrow^{may} \wedge (Q_2P_{j_1} \ldots P_{j_{l-1}})\Downarrow^{may}$ *, or*

  2. $(Q_1P_{i_1} \ldots P_{i_{k-1}})\Downarrow^{may} \wedge (Q_2P_{j_1} \ldots P_{j_l})\Downarrow^{may}$

Proof:

1. (Reverse direction) Let $0 \leq n$. Assume that
   $(\exists\langle i_1 \ldots i_k\rangle, \langle j_1 \ldots j_l\rangle$ such that

   - $\langle i_1 \ldots i_k\rangle$ and $\langle j_1 \ldots j_l\rangle$ are (possibly empty) strictly increasing sequences of integers from $\{1 \ldots n\}$

- $k + l = n$
- $(Q_1 P_{i_1} \dots P_{i_k})\Downarrow^{may} \wedge (Q_2 P_{j_1} \dots P_{j_{l-1}})\Downarrow^{may}$

(The case when $(Q_1 P_{i_1} \dots P_{i_{k-1}})\Downarrow^{may} \wedge (Q_2 P_{j_1} \dots P_{j_l})\Downarrow^{may}$ is proved by a similar argument).

The proof is by induction on n. Base case, $n = 0$ follows from part 1 of lemma 3. Assume result for $n = s$. Let $n = s + 1$. Consider $(Q_1|Q_2)P_1 \dots P_n$. Note that we have $i_1 = 1$ or $j_1 = 1$. Without loss of generality, assume that $i_1 = 1$. (The other case can be proved by an argument similar to the one below). From the assumption that $(Q_1 P_{i_1} \dots P_{i_{k-1}})\Downarrow^{may}$, it can be deduced that there is a reduction $Q_1 \xrightarrow{*} Q'_1$, where Q'_1 is of form $(M_1| \dots |\lambda\langle x_1 \dots x_g\rangle.M| \dots |M_t)$, for some $0 \leq t$, such that $(M_1| \dots |\lambda\langle x_1 \dots x_{h-1}, x_{h+1} \dots x_g\rangle.[x_i \mapsto P_{i_1}]M| \dots |M_t)P_{i_2} \dots P_{i_{k-1}}\Downarrow^{may}$.
Note that we have,

- $(Q_1|Q_2)P_1 \dots P_n \xrightarrow{*}$
  $(M_1| \dots |\lambda\langle x_1 \dots x_{h-1}, x_{h+1} \dots x_g\rangle.[x_h \mapsto P_{i_1}]M| \dots |M_t|Q_2)P_2 \dots P_n$
- $(M_1| \dots |\lambda\langle x_1 \dots x_{h-1}, x_{h+1} \dots x_g\rangle.[x_h \mapsto P_{i_1}]M| \dots |M_t)P_{i_2} \dots P_{i_k}\Downarrow^{may}$
- $(Q_2 P_{j_1} \dots P_{j_{l-1}})\Downarrow^{may}$
- $k - 1 + l = n - 1$
- $\langle i_2 \dots i_{k-1}\rangle$ and $\langle j_1 \dots j_l\rangle$ are (possibly empty) strictly increasing sequences of integers from in $\{2 \dots n\}$

So, the inductive hypothesis can be used. i.e

$$(M_1| \dots |\lambda\langle x_1 \dots x_{h-1}, x_{h+1} \dots x_g\rangle.[x_h \mapsto P_{i_1}]M| \dots |M_t|Q_2)P_2 \dots P_n\Downarrow^{may}$$

Thus, we have $(Q_1|Q_2)P_1 \dots P_n\Downarrow^{may}$.

2. (Forward direction)
   The proof proceeds by induction on r, where r is the length of the reduction $(Q_1|Q_2)P_1 \dots P_n \xrightarrow{*} M$, such that M is of form $(M_1| \dots |\lambda\langle x_1 \dots x_s\rangle.M| \dots |M_t)$. Note that the base case ($r = 0$) follows immediately. Let $(Q_1|Q_2)P_1 \dots P_n \rightarrow M'$ be the first step of the reduction sequence

$(Q_1|Q_2)P_1 \dots P_n \xrightarrow{*} M$. We have the following (mutually exclusive) cases depending on the reduction $(Q_1|Q_2)P_1 \dots P_n \rightarrow M'$.

- $Q_1 \rightarrow Q_1'$, and $M' = (Q_1'|Q_2)P_1 \dots P_n$. Result follows by the induction hypothesis on $(Q_1'|Q_2)P_1 \dots P_n$
- $Q_2 \rightarrow Q_2'$, and $M' = (Q_1|Q_2')P_1 \dots P_n$. Result follows by the induction hypothesis on $(Q_1|Q_2')P_1 \dots P_n$.
- The first step is a β reduction that involves P_1. Without loss of generality, assume that Q_1 has form $(N_1|\dots|\lambda\langle x_1 \dots x_s\rangle.N|\dots|N_g)$, for some $0 \leq g$, and the first step is
  $(N_1|\dots|\lambda\langle x_1 \dots x_s\rangle.N|\dots|N_g|Q_2)P_1 \dots P_n \rightarrow$
  $(N_1|\dots|\lambda\langle x_1 \dots x_{h-1}, x_{h+1} \dots x_s\rangle.[x_h \mapsto P_1]N|\dots|N_g|Q_2)P_2 \dots P_n$
  where $1 \leq h \leq s$. (The case in which Q_2 has this form can be handled similarly). Notice that this term satisfies the induction hypothesis. Let
  $Q_1' = (N_1|\dots|\lambda\langle x_1 \dots x_{h-1}, x_{h+1} \dots x_s\rangle.[x_h \mapsto P_1]N|\dots|N_g)$.
  So, we have
  $(\exists\langle i_1 \dots i_k\rangle, \langle j_1 \dots j_l\rangle$ such that
  - $\langle i_1 \dots i_k\rangle$ and $\langle j_1 \dots j_l\rangle$ are (possibly empty) strictly increasing sequences of integers from $\{2 \dots n\}$
  - $k + l = n - 1$
  - At least one of the following hold:
    (a) $(Q_1'P_{i_1} \dots P_{i_k})\Downarrow^{may} \wedge (Q_2P_{j_1} \dots P_{j_{l-1}})\Downarrow^{may}$, or
    (b) $(Q_1'P_{i_1} \dots P_{i_{k-1}})\Downarrow^{may} \wedge (Q_2P_{j_1} \dots P_{j_l})\Downarrow^{may}$

  . The result follows for $(Q_1|Q_2)P_1 \dots P_n$ by setting
  - The sequence for Q_1 is the sequence got by adding 1 to the sequence for Q_1'
  - The sequence for Q_2 is the same sequence as that obtained from the induction hypothesis. ∎

The must converge situation is rather like the may converge situation but is more natural to state.

Lemma 27 *Let* $0 \leq n$. *Then* $(Q_1|Q_2)P_1 \ldots P_n \Downarrow^{must} \;\Leftrightarrow$
$(\forall \langle i_1 \ldots i_k \rangle,\ \langle j_1 \ldots j_l \rangle$ *such that*

- $\langle i_1 \ldots i_k \rangle$ *and* $\langle j_1 \ldots j_l \rangle$ *are (possibly empty) strictly increasing sequences of integers from* $\{1 \ldots n\}$
- $k + l = n$
- *Both of the following hold:*
  1. $(Q_1 P_{i_1} \ldots P_{i_k})\Downarrow^{must} \wedge (Q_2 P_{j_1} \ldots P_{j_l})\Downarrow^{must}$
  2. $(Q_1 P_{i_1} \ldots P_{i_k})\Downarrow^{must} \wedge (Q_2 P_{j_1} \ldots P_{j_l})\Downarrow^{must}$

Proof:

- (Forward implication)

  Proof is by induction on n. Part 2 of lemma 3 proves the base case. Assume the result for $n = s$. Consider $n = s + 1$. Let

  - $P_1 \ldots P_n \in \Lambda_0$
  - $\langle i_1 \ldots i_k \rangle,\ \langle j_1 \ldots j_l \rangle$ be such that
    - $(Q_1|Q_2)P_1 \ldots P_n \Downarrow^{must}$
    - $\langle i_1 \ldots i_k \rangle$ and $\langle j_1 \ldots j_l \rangle$ are (possibly empty) strictly increasing sequences of integers from $\{1 \ldots n\}$
    - $k + l = n$

  Without loss of generality, assume that $i_1 = 1$. (The case in which $j_1 = 1$ can be handled similarly.) Note that
  $(Q_1|Q_2)P_1 \ldots P_n \Downarrow^{must} \Rightarrow Q_1 \Downarrow^{must} \wedge Q_2 \Downarrow^{must}$. Consider any reduction sequence
  $Q_1 \xrightarrow{*} N_1 | \ldots | \lambda\langle x_1 \ldots x_s \rangle.N | \ldots | N_g$
  Consider any possible reduction $Q_1 P_{i_1} \ldots P_{i_k} \rightarrow$
  $(N_1 | \ldots | \lambda\langle x_1 \ldots x_{h-1}, x_{h+1} \ldots x_s \rangle.[x_h \mapsto P_1]N | \ldots | N_g)P_{i_2} \ldots P_{i_k}$.
  Note that

  - $(Q_1|Q_2)P_1 \ldots P_n \xrightarrow{*}$
    $(N_1 \ldots | \lambda\langle x_1 \ldots x_{h-1}, x_{h+1} \ldots x_s \rangle.[x_h \mapsto P_1]N | \ldots N_g | Q_2)P_2 \ldots P_n$
  - $(N_1 \ldots | \lambda\langle x_1 \ldots x_{h-1}, x_{h+1} \ldots x_s \rangle[x_h \mapsto P_1]N | \ldots N_g | Q_2)P_2 \ldots P_n \Downarrow^{must}$

From the induction hypothesis,

- $(Q_2 P_{j_1} \ldots P_{j_l})\Downarrow^{must}$
- $(N_1|\ldots|\lambda\langle x_1 \ldots x_{h-1}, x_{h+1} \ldots x_s\rangle.[x_h \mapsto P_1]N|\ldots|N_g)P_{i_2} \ldots P_{i_k} \Downarrow^{must}$

The result follows, since the above argument holds for *ALL* possible reduction sequences of Q_1.

- (Reverse implication)

  Proof is by induction on n. Part 2 of this lemma proves the base case. Note that we have

  $(\forall\langle i_1 \ldots i_k\rangle,\ \langle j_1 \ldots j_l\rangle$ such that

  - $\langle i_1 \ldots i_k\rangle$ and $\langle j_1 \ldots j_l\rangle$ are (possibly empty) strictly increasing sequences of integers from $\{1 \ldots n\}$
  - $k + l = n$

  is true, both of the following hold:

  1. $(Q_1 P_{i_1} \ldots P_{i_k})\Downarrow^{must} \ \wedge (Q_2 P_{j_1} \ldots P_{j_l})\Downarrow^{must}$
  2. $(Q_1 P_{i_1} \ldots P_{i_k})\Downarrow^{must} \ \wedge (Q_2 P_{j_1} \ldots P_{j_l})\Downarrow^{must}$

  Hence, we deduce $Q_1\Downarrow^{must} \ \wedge \ Q_2\Downarrow^{must}$. Consider any reduction sequence r of $(Q_1|Q_2)P_1 \ldots P_n$. From the above remark, we note that there is an initial segment of the above reduction sequence such that $(Q_1|Q_2)P_1 \ldots P_n \xrightarrow{*}$ $(N_1|\ldots|\lambda\langle x_1 \ldots x_s\rangle.N|\ldots|N_g)P_1 \ldots P_n$, and the next term in the reduction sequence is

  $(N_1|\ldots|\lambda\langle x_1 \ldots x_{h-1}, x_{h+1} \ldots x_s\rangle.[x_h \mapsto P_1]N|\ldots|N_g)P_2 \ldots P_n$.

  Without loss of generality, assume that

  - $Q_1 \xrightarrow{*} (N_1|\ldots|\lambda\langle x_1 \ldots x_s\rangle.N|\ldots|N_f)$
  - $Q_2 \xrightarrow{*} (N_{f+1}|\ldots N_g)$

  (The symmetric case with the roles of Q_1 and Q_2 reversed can be handled similarly). From assumption of part of lemma that is being proved, we get $(\forall\langle i_1 \ldots i_k\rangle,\ \langle j_1 \ldots j_l\rangle$ such that

- $\langle i_1 \ldots i_k \rangle$ and $\langle j_1 \ldots j_l \rangle$ are (possibly empty) strictly increasing sequences of integers from $\{2 \ldots n\}$
- $k + l = n - 1$

is true, both of the following hold,

1. $(N_1| \ldots |\lambda\langle x_1 \ldots x_{h-1}, x_{h+1} \ldots x_s\rangle.[x_h \mapsto P_1]N| \ldots |N_f)P_{i_1} \ldots P_{i_k})\Downarrow^{must}$
2. $(N_{f+1}| \ldots N_g)P_{j_1} \ldots P_{j_l})\Downarrow^{must}$

Using the induction hypothesis, we have $(N_1| \ldots |\lambda\langle x_1 \ldots x_{h-1}, x_{h+1} \ldots x_s\rangle.[x_h \mapsto P_1]N| \ldots |N_g)P_2 \ldots P_n\Downarrow^{must}$. In particular, r terminates. ∎

Lemma 28 $M \preceq M' \ \wedge \ N \preceq N' \Rightarrow \ M|N \preceq M'|N'$

Proof: Follows easily from lemmas 26,27 ∎

Full proof

The idea of the proof is quite simple but the details are a little complicated since one has to keep track of redices carefully. The basic idea is as follows. Suppose that $N \preceq M$, we want to show that for any context, $C[]$, $C[N] \preceq C[M]$. Given any terms $P_1, \ldots, P_j$ we need to show that if $C[N]P_1, \ldots, P_j\Downarrow^{may}$ then $C[M]P_1, \ldots, P_j\Downarrow^{may}$ as well; there is an analogous condition with $\Downarrow^{must}$. If the reduction occur only inside the context the result is immediate. Thus what we need to keep track of is when terms are inserted into the "functional" position in a context. The structure of the proof resembles the structure of the corresponding proof for the lazy lambda calculus.

Two reductions $M \rightarrow_{\sigma'} M'$, and $M \rightarrow_{\sigma''} M''$ are different if $\sigma' \neq \sigma''$. Note that there are only finitely many different reductions. Let $M \in \Lambda_0$. Construct a tree with labelled edges corresponding to M denoted by $T(M)$ as follows. Let $M \rightarrow_{\sigma_i} M_i$, be all the possible different one step reductions from M. Then, the root has an edge for each label σ_i. The subtree at the node at the other end of the edge with label σ_i is the one obtained by doing the construction for M_i. Also, by a Konig's lemma argument, we deduce that $M\Downarrow^{must} \Rightarrow T(M)$ is finite.

Definition 36 *Let $\langle D, \leq \rangle$ be the domain of labelled, finitely-branching trees of finite depth, where the ordering relation $\leq$ is the subtree ordering.*

Note that $\langle D, \leq \rangle$ is well-founded. Furthermore, if $M{\Downarrow}^{must}$ and $M \rightarrow M'$, then $T(M') \leq T(M)$.

Define the contexts C[] with holes by the following grammar:
$C[] ::= x \;||\; [] \;||\; C_1[]|C_2[] \;||\lambda\langle x_1 \ldots x_k\rangle.C_1[] \;||(C_1[])(C_2[])$
The following definition is intended to capture the idea of a hole occurring in a "functional" position.

Definition 37 *(Functional occurrences of "holes")*

- [] *occurs functionally in* []
- [] *occurs functionally in* $C_1[]|C_2[]$ *if at least one of the following hold:*
  - [] *occurs functionally in* $C_1[]$
  - [] *occurs functionally in* $C_2[]$
- [] *occurs functionally in* $(C_1[])(C_2[])$ *if*
  [] *occurs functionally in* $C_1[]$

Lemma 29 *The contexts $D[]$ such that [] does not occur functionally in $D[]$ are generated by the following grammar:*
$D[] ::= x \;||\; D_1[]|D_2[] \;||\lambda\langle x_1 \ldots x_k\rangle C[] \;||(D[])(C[])$
where $C[]$ is ANY context at all

Proof: Structural induction ∎

Let $C[]$ be any context with a hole. Let M be any term. Then
$[[] \mapsto M]C[]$ is the term got by substituting M for [] in $C[]$, and is usually denoted by $C[M]$.

Let M be any term. We define the notion of *substituting M for the functional occurrences of* [] in $C[]$, denoted by $[[] \mapsto_f M]C[]$ by structural induction on $C[]$.

- $[[] \mapsto_f M]x = x$
- $[[] \mapsto_f M][] = M$

- $[[] \mapsto_f M](C_1[]|C_2[]) = ([[] \mapsto_f M]C_1[])|([[] \mapsto_f M]C_2[])$

- $[[] \mapsto_f M]\lambda\langle x_1 \dots x_k\rangle.C_1[] = \lambda\langle x_1 \dots x_k\rangle.C_1[]$

- $[[] \mapsto_f M](C_1[])(C_2[]) = ([[] \mapsto_f M]C_1[])(C_2[])$

Note that $[]$ does not occur functionally in $[[] \mapsto_f M]C[]$.

Lemma 30 *Let* $(\forall P \in \mathit{Terms})[([[] \mapsto_f P]D[]) \equiv ([[] \mapsto P]D[]) \in \mathit{Terms}]$. *Then,* $[(M \preceq N) \Rightarrow [[] \mapsto M]D[] \preceq [[] \mapsto M]D[]]$

Proof: Note that the hypothesis of the lemma means that $D[]$ is a member of the contexts generated by the grammar:
$E[] ::= x \mid\mid [] \mid\mid (E_1[]|E_2[]) \mid\mid \lambda\langle x_1 \dots x_k\rangle.P \mid\mid (C_1[])P$
where P is any term of the calculus. Proof now follows by structural induction. (Monotonicity of $|$ is used in a case) ∎

A symmetric notion of *substituting* M *for the non-functional occurrences of* $[]$ in $C[]$, denoted by $[[] \mapsto_{nf} M]C[]$ is defined by structural induction on $C[]$.

- $[[] \mapsto_{nf} M]x = x$

- $[[] \mapsto_{nf} M][] = []$

- $[[] \mapsto_{nf} M](C_1[]|C_2[]) = ([[] \mapsto_{nf} M]C_1[])|([[] \mapsto_{nf} M]C_2[])$

- $[[] \mapsto_{nf} M]\lambda\langle x_1 \dots x_k\rangle.C_1[] = \lambda\langle x_1 \dots x_k\rangle.[[] \mapsto M]C_1[]$

- $[[] \mapsto_{nf} M](C_1[])(C_2[]) = ([[] \mapsto_{nf} M]C_1[])([[] \mapsto M]C_2[])$

Note that
$C[M] = [[] \mapsto_f M]([[] \mapsto_{nf} M]C[]) = [[] \mapsto_{nf} M]([[] \mapsto_f M]C[])$.
Define a syntactic equality on contexts as follows:

Definition 38 *The syntactic equality* $\equiv$ *is the congruence (with respect to substitution) that is generated by the equation:*
$p|(q|r) \equiv (p|q)|r$

Define a reduction relation on contexts as follows:

- $(\lambda\langle x_1 \ldots x_k\rangle.C_1[])C_2[] \rightarrow \lambda\langle x_1 \ldots x_{i-1}, x_{i+1} \ldots x_k\rangle.[x_i \mapsto C_2[]]C_1[]$
  if $1 \leq i \leq k$

- $(C_1[]|\ldots|\lambda\langle x_1 \ldots x_k\rangle.C[]|\ldots|C_n[])C'[] \rightarrow$
  $C_1|\ldots|\lambda\langle x_1 \ldots x_{i-1}, x_{i+1} \ldots x_k\rangle.[x_i \mapsto C'[]]C[]|\ldots|C_n[]$, if $1 \leq i \leq k$

- $C_1[] \rightarrow C_1'[] \Rightarrow C_1[]|C_2[] \rightarrow C_1'[]|C_2[]$

- $C_2[] \rightarrow C_2'[] \Rightarrow C_1[]|C_2[] \rightarrow C_1[]|C_2'[]$

- $C_1[] \rightarrow C_1'[] \Rightarrow (C_1[])(C_2[]) \rightarrow (C_1'[])(C_2[])$

Lemma 31 *$[]$ does not occur functionally in $C[] \Rightarrow$
$(C[M] \rightarrow T \Rightarrow C[] \rightarrow D[] \wedge D[M] \equiv T)$*

Proof: Structural induction, and the characterization of contexts of hypothesis of lemma, as in lemma 29 . ∎

Lemma 32 $P \preceq Q \Rightarrow [(\forall C[])\ [C[P]\Downarrow^{may} \Rightarrow C[Q]\Downarrow^{may}]]$

Proof: Proof proceeds by induction on the length n of the reduction $C[P] \xrightarrow{*} (N_1|\ldots|\lambda\langle x_1 \ldots x_s\rangle.N|\ldots|N_f)$. Note that the case $n = 0$ is immediate. Assume the result for $n = s$. We have the following two (mutually exclusive) cases.

- ($[]$ does not occur functionally in $C[]$).
  From lemma 31, $C[M] \rightarrow T \Rightarrow C[] \rightarrow D[] \wedge D[M] \equiv T$, . Since $D[M]\Downarrow^{may}$ in s steps, from the induction hypothesis, $D[N]\Downarrow^{may}$. Since $C[N] \rightarrow D[N]$, $C[N]\Downarrow^{may}$.

- ($[]$ occurs functionally in $C[]$)
  Define $D[] = [[] \mapsto_f M]C[]$. Note that $D[M] = C[M]$. Since, $D[M]\Downarrow^{may}$, from case the above $D[N]\Downarrow^{may}$. Note that
  $D[N] = [[] \mapsto_{nf} N]([[] \mapsto_f M]C[]) = [[] \mapsto_f M]([[] \mapsto_{nf} N]C[])$
  From lemma 30, we deduce that
  $C[N] = [[] \mapsto_f N]([[] \mapsto_{nf} N]C[])\Downarrow^{may}$ ∎.

Lemma 33 $P \preceq Q \Rightarrow [(\forall C[])\ [C[P]\Downarrow^{must} \Rightarrow C[Q]\Downarrow^{must}]]$

Proof: Proof proceeds by induction on $T(C[M])$. The base case is immediate. For the induction step, we have the following two (mutually exclusive) cases.

- ([] does not occur functionally in $C[]$).
  Then, we have $C[N]{\rightarrow}T{\Rightarrow}\ C[]{\rightarrow}D[]\wedge\ D[N]\ \equiv\ T$ from lemma 31. So $C[M]{\rightarrow}D[M]$. Since $D[M]{\Downarrow}^{must}$ and $T(D[M]) \leq T(C[M])$, the induction hypothesis can be used to deduce $D[N]{\Downarrow}^{must}$. This is true for any reduction of $C[N]$. Hence, $C[N]{\Downarrow}^{must}$.

- ([] occurs functionally in $C[]$
  Define $D[] = [[] \mapsto_f M]C[]$. Note that $D[M] = C[M]$. Since, $D[M]{\Downarrow}^{must}$, from case the above $D[N]{\Downarrow}^{must}$. Note that
  $D[N] = \ [[] \mapsto_{nf} N]([[] \mapsto_f M]C[]) = \ [[] \mapsto_f M]([[] \mapsto_{nf} N]C[])$
  From lemma 30, we deduce that
  $C[N] = \ [[] \mapsto_f N]([[] \mapsto_{nf} N]C[]){\Downarrow}^{must}$ ∎

With these lemmas in hand the proof of operational extensionality is complete.

Theorem 1 *(Operational extensionality)* $M{\preceq}N \Leftrightarrow (\forall C[.])\,[C[M]{\preceq}C[N]]$

Proof: From lemmas 32 and 33, $M{\preceq}N \Leftrightarrow (\forall C[.])\,[C[M]{\preceq_0}C[N]]$. Clearly, any terms that we wish to use as arguments to $C[]$ can be absorbed into another context $D[]$. Thus $M{\preceq}N \Leftrightarrow (\forall C[.])\,[C[M]{\preceq}C[N]]$. ∎

3.4 The Powerdomain Construction

In this section we define the powerdomain construction that we use. We introduce it as a functor in a certain category of nondeterministic continuous algebras. We obtain a model of the γ-calculus by constructing a solution to a recursive domain equation in the usual way [65].

Many of the ideas are the same as in the analysis of the lazy λ-calculus but the details are somewhat more complicated. Before we begin with the mathematical details we discuss some motivational issues. As the adequacy proof shows, semantic equality in our model is at least as fine as the equality induced by the operational preorder. In fact, the same proof also shows that one can construct an adequate model for the fragment of the γ-calculus that we consider using the Plotkin powerdomain [52]. Why, then, did we choose to use this powerdomain rather than Plotkin's?

Our model is probably not fully abstract but it is, in some sense, "closer to being fully abstract" than a model based on the Plotkin powerdomain would be. In order to clarify this point, consider the example discussed at the end of the previous section. The terms $\lambda x.[\Omega \ or \ \lambda y.\Omega]$ and $\lambda xy.\Omega \ or \ \lambda x.\Omega$ were deemed equivalent by the operational semantics. In order to avoid confusion we use the notation $up(x \mapsto e)$ to represent lifted functions in the semantic domains. Intuitively, we expect the term Ω to denote $\perp$ in the semantic model. Thus the denotations of the pair of terms above are $\{up(x \mapsto \{\perp, up(y \mapsto \perp)\})$ and $\{up(x \mapsto \perp), up(y \mapsto up(x \mapsto \perp))\}$. Is is easy to check that these are not Egli-Milner related. Thus, a model based on the Plotkin powerdomain would not identify these terms. The difference arises from the way we order the finite sets. We do not use the Egli-Milner order, rather we use the fact that we have sets of functions and use an order that reflects the applicative behavior of the sets.

The intuition is that the partial order of the domain must satisfy the defining equations of the operational preorder. This idea can be treated formally by introducing generalized versions of quasi-applicative transition systems used in the study of the lazy lambda calculus [4,50]. We restrict ourselves to an informal discussion. Define semantic versions of the convergence predicates as follows:

- $f{\downarrow}^{\mathbf{may}}$ if $f \neq \perp$
- $f{\downarrow}^{\mathbf{must}}$ if $\perp \notin f$

From lemma 24, the operational preorder $\preceq$ satisfies :

$$M \preceq N \Leftrightarrow \begin{cases} M{\Downarrow}^{must} \Rightarrow N{\Downarrow}^{must} \\ M{\Downarrow}^{may} \Rightarrow N{\Downarrow}^{may} \\ (\forall P) MP \preceq NP \end{cases}$$

Let $\diamond$ be notation for the semantic application function. Thus we would like the partial order of the domain to satisfy:

$$f \sqsubseteq g \Leftrightarrow \begin{cases} f{\downarrow}^{\mathbf{may}} \Rightarrow g{\downarrow}^{\mathbf{may}} \\ f{\downarrow}^{\mathbf{must}} \Rightarrow g{\downarrow}^{\mathbf{must}} \\ \forall x. f \diamond x \sqsubseteq g \diamond x \end{cases}$$

The above holds in our model but not in the model based on the Plotkin powerdomain. In the Plotkin powerdomain, only the left to right implication holds. This point can be clarified further. Define a preorder $\leq$ on the elements of the semantic domain as follows. The definition mimics the definition 35 of the operational preorder $\preceq$

- $f \leq_0 g$ if
  - $f\downarrow^{\mathbf{may}} \Rightarrow g\downarrow^{\mathbf{may}}$
  - $f\downarrow^{\mathbf{must}} \Rightarrow g\downarrow^{\mathbf{must}}$
- $f \leq_{k+1} g$ if
  - $f \leq_k g$
  - $(\forall x)\, [f \diamond x \leq_k g \diamond x]$
- $\leq = \bigcap \leq_k,\ k \in \omega$

In our model, $\leq$ coincides with the partial ordering of the domain $\sqsubseteq$. In the model based on Plotkin powerdomain, the domain ordering $\sqsubseteq$ is a *strict* refinement of the ordering $\leq$.

Basic notation

All domains in this section are SFP objects. We use $B(D)$ as notation for the basis of D. We follow the notation of the work of S. Abramsky and L. Ong on the lazy lambda calculus [4]. Recall the definition of lifting as the left adjoint of the forgetful functor from $CPO_\perp$ to CPO where $CPO_\perp$ is the subcategory of strict functions. Let D, E objects of CPO. Let $f \in D \to E$.

- $D_\perp$ is the cpo defined as follows:
  - $|D_\perp| = \{\perp\} \bigcup \{\langle 0, d\rangle | d \in D\}$
  - Let $y,\ z \in D_\perp$. Then
    $y \sqsubseteq z \Leftrightarrow y = \perp \ \vee\ [y = \langle 0, d_1\rangle \ \wedge\ z = \langle 0, d_2\rangle \wedge\ d_1 \sqsubseteq_D d_2$
- If $d \in D$, define $up(d) = \langle 0, d\rangle$
- Define $lift(f) \in D_\perp \to E$ by:
  - $lift(f)\,(\perp) = \perp_E$
  - $lift(f)\,(\langle 0, d\rangle) = f\,(d)$
- Let $dn_D = lift(id_D)$

- $\diamond : (D_1 \to D_2)_\perp \times D_1 \to D_2$ is defined by
  - $\perp \diamond x = \perp$
  - $up(f) \diamond x = f(x)$, where $f \in D_1 \to D_2$

Definition 39 *$\langle D, \star \rangle$ is a continuous algebra if $\star$ is a continuous function from $D \times D$ to D, satisfying upper semi-lattice axioms*

Definition 40 *Let $\langle D_1, \star_1 \rangle$ and $\langle D_2, \star_2 \rangle$ be continuous algebras. Let $f \in D_1 \to D_2$. f is said to be* **linear** *if*
$(\forall \{x_1, x_2\} \subseteq D_1)\, [f(x_1 \star_1 x_2) = f(x_1) \star_2 f(x_2)]$

Definition 41 *Let $\langle D_1, \star_1 \rangle$ and $\langle D_2, \star_2 \rangle$ be continuous algebras. Then, (e, p) is a linear embedding-projection pair if the following hold:*

- $(p \circ e) = 1_{D_1}$
- $(e \circ p) \sqsubseteq 1_{D_2}$
- *e is linear*
- *p is linear*

Given any continuous algebra D, If $s = \{f_1 \ldots f_n\}$ where $(\forall i)\ [1 \leq i \leq n]\ [f_i \in (D \to D)_\perp]$, and $x \in D$ then $s \diamond x$ is notation for $f_1 \diamond x \star f_2 \diamond x \ldots \star f_n \diamond x$.

3.4.1 The Powerdomain Functor

In this subsection we define the powerdomain functor and show that it is continuous on a category of algebras very closely related to the bifinites (SFP).

Definition 42 *Let $\langle D, \star \rangle$ be a continuous algebra. Then $P(D)$ is a preorder defined as follows:*

- $|P(D)| = \{s | s \in P_{fin}(B((D \to D)_\perp))$
- $s_1 \sqsubseteq s_2 \Leftrightarrow$
  1. $\perp \in s_2 \Rightarrow \perp \in s_1$
  2. $s_1 \neq \{\perp\} \Rightarrow s_2 \neq \{\perp\}$
  3. $(\forall x \in D)[s_1 \diamond x \sqsubseteq s_2 \diamond x]$

$\overline{P}(D)$ is the ideal completion of $P(D)$. We now define a union operation on $\overline{P}(D)$ to make it a continuous algebra. Define $\uplus$ from $P(D) \times P(D)$ to $P(D)$ by $s_1 \uplus s_2 = s_1 \bigcup s_2$

Lemma 34 *$\uplus$ is monotone in each argument.*

Proof: Conditions 1 and 2 in the definition 42 of the preordering relation are easy to check. Condition 3 follows by noting that $(s_1 \uplus s_2) \diamond x = (s_1 \diamond x) \star (s_2 \diamond x)$, and from the monotonicity of $\star$. ∎

So $\uplus$ can be extended to a continuous function from $\overline{P}(D) \times \overline{P}(D)$ to $\overline{P}(D)$. The upper semi-lattice axioms are easy to check for the members of $P(D)$ and the continuity of $\uplus$ enables us to verify the laws for members of $\overline{P}(D)$.

We hope to solve the recursive domain equation $D \simeq \overline{P}(D)$. So, we need to establish a suitable category in which the above construction generalizes to a functor preserving colimits of ω- chains. Define the category $NSFP$ as follows:

- Objects:

  The objects are continuous algebras expressible as the colimits of ω-chains of finite continuous algebras, where the arrows of the chain are linear embedding-projection pairs.

- Arrows:

  The arrows are linear embedding projection pairs.

In particular, all the objects are SFP objects. The above category can be viewed intuitively as that obtained by adding colimits of countable directed diagrams of finite continuous algebras, where the arrows of the diagram are linear embedding-projection pairs. Also note that the category is a subcategory of SFP^{ep} that contains the image of the Plotkin-powerdomain functor acting on SFP^{ep}, where SFP^{ep} is the category of SFP objects with arrows embedding-projection pairs.

Lemma 35 *(Existence of colimits)*

- *$NSFP$ is closed under colimits of countable directed diagrams*
- *The one element domain is the initial object*

The recursive domain equation $D \simeq \overline{P}(D)$ can be solved in $NSFP$ if we can prove that $\overline{P}(.)$ is a functor on $NSFP$ that preserves colimits of ω-chains. We now define the action of $\overline{P}(.)$ on linear embedding projection pairs. Let $\langle D_1, \star_1\rangle$ and $\langle D_2, \star_2\rangle$ be continuous algebras. Let (e, p), be a linear embedding-projection pair. Define e' as follows:

- Define $e' : P(D_1) \rightarrow \overline{P}(D_2)$ by:
  - $e'(\bot) = \bot$
  - $e'(\langle 0, f\rangle) = up\,(((e \circ dn\,(f)) \circ p))$
  - $e'\{F_1 \ldots F_n\} = \{e'(F_1) \ldots e'(F_n)\}$

We need to show that e' is well-defined and monotone.

Lemma 36 *Let $s_1 = \{F_1 \ldots F_n\}$ and $s_2 = \{H_1 \ldots H_m\}$ be elements of $P(D_1)$, such that $s_1 \sqsubseteq s_2$. Then $e'(s_1) \sqsubseteq e'(s_2)$.*

Proof: We check the three conditions of Definition 42.

1. $\bot \in e'(s_2)$
   $\Rightarrow \bot \in s_2$ [from definition of e']
   $\Rightarrow \bot \in s_1$ [as $s_1 \sqsubseteq s_2$]
   $\Rightarrow \bot \in e'(s_1)$

2. $\{\bot\} \neq e'(s_1)$
   $\Rightarrow \{\bot\} \neq s_1$ [from definition of e']
   $\Rightarrow \{\bot\} \neq s_2$ [as $s_1 \sqsubseteq s_2$]
   $\Rightarrow \{\bot\} \neq e'(s_2)$

3. Let $x \in D_2$. Using the linearity of e, it follows that $e'(s_1) \diamond x = e(s_1 \diamond p(x))$. Similarly, we deduce $e'(s_2) \diamond x = e(s_2 \diamond p(x))$. $s_1 \sqsubseteq s_2 \Rightarrow s_1 \diamond p(x) \sqsubseteq s_2 \diamond p(x)$. The result follows from the monotonicity of e. ∎

Thus e' is well-defined, monotone and extends uniquely to a continuous function from $\overline{P}(D_1)$ to $\overline{P}(D_2)$. Furthermore, it follows immediately from the definition of e' that $e'(s_1 \uplus s_2) = e'(s_1) \uplus e'(s_2)$, for $s_1, s_2 \in P(D_1)$. The result, for arbitrary elements of $\overline{P}(D_1)$, follows from the continuity of all the functions involved.

The situation for p' is almost identical. Define p' as follows:

- $p'(\perp) = \perp$
- $p'(\langle 0, g\rangle) = up\ (((p \circ dn\ (g)) \circ e))$
- $p'\{F_1 \ldots F_n\} = \{p'(F_1) \ldots p'(F_n)\}$

Lemma 37 *Let t_1 and t_2 be elements of $P(D_2)$, such that $t_1 \sqsubseteq t_2$. Then $p'(t_1) \sqsubseteq p'(t_2)$.*

Proof: Similar to the previous lemma. ∎

Thus p' is also well-defined, monotone and extends uniquely to a continuous function from $\overline{P}(D_2)$ to $\overline{P}(D_1)$. Also, it follows immediately from the definition of p' that $p'(t_1 \uplus t_2) = p'(t_1) \uplus p'(t_2)$, for $t_1, t_2 \in P(D_2)$. The result, for arbitrary elements of $\overline{P}(D_2)$, follows from the continuity of all functions involved.

Since e', p' are linear and continuous, the proof that (e', p') is a linear embedding projection pair reduces to the following lemma.

Lemma 38 *Let $F \in B((D_1 \to D_1)_\perp)$, $G \in B((D_2 \to D_2)_\perp$. Then,*

- $(p' \circ e')(F) = F$
- $(e' \circ p')(G) \sqsubseteq G$

Proof:

1. Proving that $(p' \circ e')(F) = F$:
   - $(p' \circ e')(\perp) = \perp$
   - $(p' \circ e')(\{\langle 0, f\rangle\}) = p'(\{up\ (((p \circ f) \circ e))\}$
     $= up\ (((p \circ ((e \circ f) \circ p)) \circ e)) = \langle 0, f\rangle$ [as $(p \circ e) = id_{D_1}$]
2. Proving that $(e' \circ p')(G) \sqsubseteq G$:
   Similar to above but using $(p \circ e) \sqsubseteq id_{D_2}$ ∎

Now we have the machinery to define the action of the functor on the morphisms of the category $NSFP$. Define $\overline{P}((e, p)) = (e', p')$. It is easy to check that

- $\overline{P}((id_D, id_D)) = (id_D, id_D)$, for any continuous algebra $\langle D, \star\rangle$

- Let (e_1, p_1) be a linear embedding projection pair between $\langle D_1, \star_1\rangle$, $\langle D_2, \star_2\rangle$. Let (e_2, p_2) be a linear embedding projection pair between $\langle D_2, \star_2\rangle$, $\langle D_3, \star_3\rangle$. $((e_1 \circ e_2), (p_2 \circ p_1))$ is a linear embedding projection pair between $\langle D_1, \star_1\rangle$ and $\langle D_3, \star_3\rangle$ and $\overline{P}((e_2 \circ e_1), (p_1 \circ p_2)) = (\overline{P}((e_2, p_2)) \circ \overline{P}((e_1, p_1)))$

The final lemma establishes that this functor is continuous and thus one can solve recursive domain equations using it.

Lemma 39 *Let* $\Delta = \langle D_m, \langle f_{mn}, f_{nm}\rangle\rangle$ *be a chain of linear embedding projection pairs. Let* $\langle D, \rho\rangle = \underrightarrow{Colim}\ \Delta$. *Then,* $\underrightarrow{Colim}\ \overline{P}(\Delta) \simeq \overline{P}(\underrightarrow{Colim}\ \Delta)$, *where we write* $\underrightarrow{Colim}\ \overline{P}(\Delta)$ *for* $\underrightarrow{Colim}\ \langle \overline{P}(D_m), \overline{P}(\langle f_{mn}, f_{nm}\rangle)\rangle$.

Proof: It suffices (lemma 2, [51][ch 4, page 11]) to check that $\bigsqcup(\overline{P}(\rho_n) \circ \overline{P}(\rho_n^R)) = id_{\overline{P}(D)}$. From the linearity and continuity of $\overline{P}(\rho_n)$ and $\overline{P}(\rho_n^R)$ it suffices to check $\bigsqcup(\overline{P}(\rho_n) \circ \overline{P}(\rho_n^R))(s) = s$, for singleton sets of $P(D)$. When we look at the singleton sets, however, it is clear that we can mimic the standard verifications of this fact [65]. ∎

3.4.2 The Model and its Basic Properties

In this section we define the model and prove some basic properties of the model. The properties are essentially tools that show how one can define structures on the domain by induction using the iterates and also how one can use the projections onto the iterates to get a handle on the finite approximants to the elements. They are similar in spirit to the "index" calculations outlined in Wadsworth's discussion of D_∞, and to the "index" calculations outlined in the discussion of the lazy lambda-calculus [4].

What is a Model of the γ-calculus?

Before constructing the initial solution we sketch how this is used to provide a model of our subset of the γ-calculus. The recursive domain equation that we solve is

$$D = \overline{P}(D).$$

We solve this equation in the category NSFP.

From an algebraic point of view we have a cpo with three continuous operations, application $\diamond$, union $\star$ and product, $\times$. These operations obey the following laws:

1. $\perp \diamond x = \perp$

2. $(d \star e) \diamond x = (d \diamond x) \star (e \diamond x)$

3. $\times$ is associative

4. $\times$ is commutative

5. $d \times \perp = d \star \perp$

6. $(d \times e) \diamond x = ((d \diamond x) \times e) \star (d \times (e \diamond x))$

7. $d \times (e \star f) = (d \times e) \star (d \times f)$.

We now have enough structure to give semantics to the fragment of the language that we are considering. The following definition uses the familiar environment mechanism. The functions Gr and Fun map between $\overline{P}(D)$ and D.

- $[\![x]\!]\ \rho\ = \rho(x)$
- $[\![\lambda x.M]\!]\ \rho\ = Gr(d \mapsto [\![M]\!]\ \rho[x \mapsto d]\ )$
- $[\![\lambda\langle x_1, x_2\rangle M]\!]\ \rho\ =$
  $Gr(\star[(d_1 \mapsto Gr(d_2 \mapsto [\![M]\!]\ \rho[x_1 \mapsto d_1, x_2 \mapsto d_2]\ )),$
  $(d_1 \mapsto Gr(d_2 \mapsto [\![M]\!]\ \rho[x_2 \mapsto d_1, x_1 \mapsto d_2]\ ))])$
- $[\![MN]\!]\ \rho\ = [\![M]\!]\ \rho\ \diamond [\![N]\!]\ \rho$
- $[\![M|N]\!]\ \rho\ = [\![M]\!]\ \rho\ \times [\![N]\!]\ \rho$

Construction of the Initial Solution

Let $\langle D_1, \star_1\rangle$ and $\langle D_2, \star_2\rangle$ be continuous algebras. Let (e, p) be a linear embedding-projection pair. Define e' as follows:

- Define $e' : B(\overline{P}(D_1)) \rightarrow \overline{P}(D_2)$ by:
  - $e'\,(\perp) = \perp$
  - $e'\,(\langle 0, f\rangle) = up\,(((e \circ dn\,(f)) \circ p))$
  - $e'\{F_1 \ldots F_n\} = \{e'(F_1) \ldots e'(F_n)\}$

$e^\star$ is the unique continuous extension of e'.

Define $p' : B(\overline{P}(D_2)) \to \overline{P}(D_1)$ as follows:

- $p'(\perp) = \perp$
- $p'(\langle 0, g\rangle) = up\,(((p \circ dn\,(g)) \circ e))$
- $p'\{f_1 \ldots f_n\} = \{p'(f_1) \ldots p'(f_n)\}$

$p^\star$ is the unique continuous extension of p'.

Let D_0 be the the one point continuous algebra, and let $D_1 = \overline{P}(D)$. Let $i_0 : D_0 \to D_1$ be defined by $i_0(\perp_0) = \perp_1$. Let $j_0 : D_1 \to D_0$ be defined by $j_0(x) = \perp_0$. Define inductively:

- $D_{n+1} = \overline{P}(D_n)$
- $\langle i_{n+1}, j_{n+1}\rangle = \langle i_n^\star, j_n^\star\rangle$

Note that $(\forall n)$ $[\langle i_n, j_n\rangle$ is a linear ep pair$]$. Then $\langle D_n, j_n\rangle_{n\in\omega}$ is an inverse system of finite continuous algebras. Define, standardly, $\phi_{m,n} : D_n \to D_m$ by

- $\phi_{n,n} = 1_{D_n}$
- $\phi_{m+1,n} = (\phi_{m,n} \circ j_m)$, if $(n \leq m,\ n \neq m)$
- $\phi_{m,n+1} = (i_n \circ \phi_{m,n})$, if $(m \leq n,\ n \neq m)$

Note that
$(\forall n, m)$ $[m \leq n \Rightarrow \langle \phi_{m,n}, \phi_{n,m}\rangle$ is a linear ep pair$]$. Identify the initial solution $D \subseteq \Pi_{n\in\omega} D_n$, as $D = \{\langle x_n\rangle_{n\in\omega} \mid x_n \in D_n \wedge j_n(x_{n+1}) = x_n\}$
ordered pointwise. Note that we have the linear ep pairs $\langle \phi_{m,\infty}, \phi_{\infty,m}\rangle$. Also, note that $\langle D, \star\rangle$ is a continuous algebra, where $\star$ is defined by
$\langle x_n\rangle_{n\in\omega} \star \langle y_n\rangle_{n\in\omega} = \langle x_n \star_n y_n\rangle_{n\in\omega}$
We write x_m for $\phi_{\infty,m}(x)$. Henceforth, we identify the element x of D_m with $\phi_{m,\infty}(x)$.

Index calculations

The proof of the following two lemmas is standard and is omitted.

Lemma 40 *Let $x \in D$. Then,*

1. $x \in D_n \Rightarrow x_n = x$

2. $x \in D_n \Rightarrow i_n(x) = x$

3. $x \in D_{n+1} \Rightarrow j_n(x) \sqsubseteq x$

Lemma 41 *Let $x \in D$. Then,*

1. $(x_n)_m = x_{min(n,m)}$

2. $n \leq m \Rightarrow x_n \sqsubseteq x_m \sqsubseteq x$

3. $x = \bigsqcup_n x_n$

4. $x \in D_n \Rightarrow (\forall m \geq n)\ [x_m = x]$

5. $\perp_n$ *is the least element of* D_n

6. $\perp_n = \perp$

Applicative behavior

Define $App_n : (D_n \to D_n)_\perp \times D_n \to D_n$ as,

- $App_n(\perp, x) = \perp$
- $App_n(up(f), x) = f(x)$, where $f \in D_n \to D_n$

Define $App_n : D_{n+1} \times D_n \to D_n$ as the left linear extension of the above. More formally, let $s = \{f_1 \ldots f_n\} \in D_{n+1}$. Then,
$App_n(s, x) = App_n(f_1, x) \star_n App_n(f_2, x) \ldots \star_n App_n(f_n, x)$. It follows from the definitions that $App_n : D_{n+1} \times D_n \to D_n$ is a monotone (and hence continuous) function.

Lemma 42 *Let $n \leq k$. Then,*

1. $App_n(x_{n+1}, y_n) \sqsubseteq App_k(x_{k+1}, y_k)$

2. $App_n(x_{n+1}, y_n) = App_k((x_{n+1})_{k+1}, y_k)$

3. $App_n(x_{n+1}, y_n) = [App_k(x_{k+1}, (y_n)_k]_n$

Proof:

1. (Proof is by induction on k)

   Consider $k = n + 1$. Depending on the structure of x_{n+2}, there are two cases:

- x_{n+2} is a singleton, i.e $x_{n+2} = \{f\}$, for some $f \in (D_n \to D_n)_\perp$. This splits up further into two cases.
  - $f = \perp$. Then $x_{n+1} = \{\perp\}$, and result follows.
  - $f = up(g)$, for some $g \in D_n \to D_n$. Then, note that $x_{n+1} = up(g')$, for some $g' \in D_{n-1} \to D_{n-1}$. So, we have

$$\begin{aligned}
App_n(x_{n+1}, y_n) &= App_n(j_{n+1}(x_{n+2}), j_n(y_{n+1})) \\
&= App_n(up((j_n \circ (dn(x_{n+2}) \circ i_n))), j_n(y_{n+1})) \\
&= ((j_n \circ (dn(x_{n+2}) \circ i_n)) \circ j_n)(y_{n+1}) \\
&\sqsubseteq (j_n \circ dn(x_{n+2}))(y_{n+1}) \\
&\sqsubseteq dn(x_{n+2})(y_{n+1}) \\
&= App_{n+1}(x_{n+2}, y_{n+1})
\end{aligned}$$

- $x_{n+2} = \{f_1 \ldots f_m\}$. Note that
  $j_{n+1}(x_{n+2}) = \{j_{n+1}(f_1) \ldots j_{n+1}(f_m)\} = \star_{n+1}\{j_{n+1}(f_1) \ldots j_{n+1}(f_m)\}$.

$$\begin{aligned}
App_n(x_{n+1}, y_n) &= App_n(j_{n+1}(x_{n+2}), j_n(y_{n+1})) \\
&= App_n(\star_{n+1}\{j_{n+1}(f_1) \ldots j_{n+1}(f_m)\}, j_n(y_{n+1})) \\
&= \star_n\{App_n(j_{n+1}(f_i), j_n(y_{n+1})) | i = 1 \ldots m\}
\end{aligned}$$

  But, from the preceding case, $App_n((f_i)_{n+1}, y_n) \sqsubseteq App_n(f_i, y_{n+1})$. Result now follows from monotonicity of $\star_n$

2. (Proof is by induction on k)
   Assume result for k. Consider $k+1$. Depending on the structure of $(x_{n+1})_{k+2}$, there are two cases:

   - $(x_{n+1})_{k+2} = \{f\}$, for some $f \in (D_{k+1} \to D_{k+1})_\perp$. This splits up further into two cases.
     - $f = \perp$. Then $x_{n+1} = \{\perp\}$, and result follows.
     - $f = up(g)$, for some $g \in D_{k+1} \to D_{k+1}$. We have

$$\begin{aligned}
App_{k+1}((x_{n+1})_{k+2}, y_{k+1}) &= App_{k+1}(i_{k+1}(x_{n+1})_{k+1}, y_{k+1}) \\
&= (i_k \circ (dn(x_{n+1})_{k+1} \circ j_k))(y_{k+1})
\end{aligned}$$

$$
\begin{aligned}
&= (i_k \circ dn(x_{n+1})_{k+1})(y_k) \\
&= i_k(App_k((x_{n+1})_{k+1}, y_k)) \\
&= i_k(App_n(x_{n+1}, y_n)) \\
&= App_n(x_{n+1}, y_n)
\end{aligned}
$$

- $(x_{n+1})_{k+2} = \{(f_1)_{k+2} \dots (f_m)_{k+2}\}$. We have

  $$App_{k+1}((x_{n+1})_{k+2}, y_{k+1}) = \star_{k+1}\{App_{k+1}((f_i)_{k+2}, y_{k+1}) | i = 1 \dots m\}$$

  From previous cases, we have

  $(\forall 1 \leq i \leq m)\ [App_{k+1}((f_i)_{k+2}, y_{k+1}) = App_n((f_i)_{n+1}, y_n)$

  So, we have

  $$
  \begin{aligned}
  App_{k+1}((x_{n+1})_{k+2}, y_{k+1}) &= \star_{k+1}\{App_{k+1}((f_i)_{k+2}, y_{k+1}) | i = 1 \dots m\} \\
  &= \star_{k+1}\{App_n((f_i)_{n+1}, y_n) | i = 1 \dots m\} \\
  &= \star_n\{App_n((f_i)_{n+1}, y_n) | i = 1 \dots m\} \\
  &= App_n(\{(f_1)_{n+1} \dots (f_m)_{n+1}\}, y_n) \\
  &= App_n(x_{n+1}, y_n) \qquad \blacksquare
  \end{aligned}
  $$

3. (Proof by induction on k)

   Assume result for k. Consider $k+1$. Depending on the structure of x_{k+2}, there are two cases:

   - x_{k+2} is a singleton, i.e $x_{k+2} = \{f\}$, for some $f \in (D_{k+1} \to D_{k+1})_\perp$. This splits up further into two cases.

     - $f = \perp$. Then $x_{n+1} = \{\perp\}$, and result follows.
     - $f = up(g)$, for some $g \in D_{k+1} \to D_{k+1}$. We have

     $$
     \begin{aligned}
     [App_{k+1}(x_{k+2}, (y_n)_{k+1})]_n &= \Phi_{k+1,n}[App_{k+1}(x_{k+2}, (y_n)_{k+1})] \\
     &= (\Phi_{k,n} \circ j_k)[App_{k+1}(x_{k+2}, (y_n)_{k+1})] \\
     &= (\Phi_{k,n} \circ (j_k \circ (dn(x_{k+2}) \circ i_k)))((y_n)_k) \\
     &= \Phi_{k,n}[App_k(j_{k+1}(x_{k+2}), (y_n)_k)] \\
     &= [App_k(j_{k+1}(x_{k+2}), (y_n)_k)]_n \\
     &= [App_k(x_{k+1}, (y_n)_k)]_n \\
     &= App_n(x_{n+1}, y_n)
     \end{aligned}
     $$

- $(x_{n+1})_{k+2} = \{(f_1)_{k+2} \cdots (f_m)_{k+2}\}$.

$$\begin{aligned}
[App_{k+1}((x_{k+2}),(y_n)_{k+1})]_n &= \Phi_{k+1,n}[App_{k+1}(x_{k+2},(y_n)_{k+1})] \\
&= [\star_{k+1}\{App_{k+1}((f_i)_{k+2}, y_{k+1})|i\}]_n \\
&= \star_n\{[App_{k+1}((f_i)_{k+2}, y_{k+1})]_n\} \\
&= \star_n[\{App_n((f_i)_{n+1}, y_n)\}] \\
&= App_n(\{(f_1)_{n+1}, \ldots (f_m)_{n+1}\}, y_n) \\
&= App_n(x_{n+1}, y_n) \qquad \blacksquare
\end{aligned}$$

Define $\diamond : D \times D \to D$ by $x \diamond y = \bigsqcup_{n\in\omega}[Ap_n(x_{n+1}, y_n)]$. Lemma 42.1 proves that the terms whose $\bigsqcup$ is being taken do form a chain.

Lemma 43 *(Coherence) Let $x \in D_{n+1}, y \in D_n$. Then,*
$x \diamond y = Ap_n(x_{n+1}, y_n)$

Proof:

$$\begin{aligned}
x \diamond y &= x_{n+1} \diamond y_n \\
&= \textstyle\bigsqcup_{i\in\omega}[Ap_i((x_{n+1})_{i+1}, (y_n)_i)] \text{ [from 42 .2]} \\
&= \textstyle\bigsqcup_{i\leq n}[Ap_i((x_{n+1})_{i+1}, (y_n)_i)] \\
&= Ap_n(x_{n+1}, y_n) \qquad \blacksquare
\end{aligned}$$

It is easy to check that $\diamond$ is a continuous function.

Lemma 44 *Let $x, y \in D$. Then,*

1. $x_{n+1} \diamond y = x_{n+1} \diamond y_n = [x \diamond y_n]_n$
2. $x_1 \diamond y = \perp$
3. $x_0 \diamond y = \perp$

Proof: Proofs of 2, 3 are omitted. 1 is proved below.

$$\begin{aligned}
x_{n+1} \diamond y &= \textstyle\bigsqcup_{i\in\omega}[App_i((x_{n+1})_{i+1}, y_i)] \text{ use 42 2} \\
&= \textstyle\bigsqcup_{i\leq n}[App_i((x_{n+1})_{i+1}, y_i)] \text{ use 42 1} \\
&= App_n(x_{n+1}, y_n)
\end{aligned}$$

$$
\begin{aligned}
[x \diamond y_n]_n &= [\bigsqcup_{i\in\omega}[App_i((x_{n+1})_{i+1}, y_i)]]_n \\
&= \bigsqcup_{i\in\omega}[App_i((x_{n+1})_{i+1}, y_i)]_n \text{ from } 42\ 3 \\
&= \bigsqcup_{i\le n}[App_i((x_{n+1})_{i+1}, y_i)]_n \\
&= \bigsqcup_{i\le n}[App_i(x_{i+1}, y_i)]_n \\
&= \bigsqcup_{i\le n}[App_i(x_{i+1}, y_i)]_n \text{ from } 42\ 1 \\
&= App_n(x_{n+1}, y_n) \\
&= x_{n+1} \diamond y_n \qquad \blacksquare
\end{aligned}
$$

Isomorphism between D **and** $\overline{P}(D)$

$\langle D, \star\rangle$ is a continuous algebra. Define $App' : (D \to D)_\perp \times D \to D$ as,

- $App'(\perp, x) = \perp$
- $App'(up(f), x) = f(x)$, where $f \in D \to D$

Define $App : P(D) \times D \to D$ as the left linear extension of App'. Formally, let $s = \{f_1 \ldots f_n\} \in D_{n+1}$. Then, $App(s, x) = App(f_1, x) \star App(f_2, x) \ldots \star App(f_n, x)$. Recall that $P(D)$ was defined as:

- $|P(D)| = \{s | s \in P_{fin}(B((D \to D)_\perp))$
- $s_1 \sqsubseteq s_2 \Leftrightarrow$
  1. $\perp \in s_2 \Rightarrow \perp \in s_1$
  2. $s_1 \neq \{\perp\} \Rightarrow s_2 \neq \{\perp\}$
  3. $(\forall x \in D)[App(s_1, x) \sqsubseteq App(s_2, x)]$

It follows from the definitions that $App : P(D) \times D \to D$ is a monotone function. Note that $B(\overline{P}(D)) = P(D)$. Extend App continuously to the whole of $\overline{P}(D)$.

Define $[rep()]_n : P(D) \to D_{n+1}$ as follows. Let $s \in P(D)$.

- s is a singleton. This splits up into two cases.
  - $s = \{\perp\}$. Define $[rep(s)]_n = \perp_{n+1}$.
  - $s = \{up(f)\}$, where $f \in B(D \to D)$. Define $[rep(s)]_n = up(\lambda y \in D_n.[App(s, y)]_n)$

- $s = \{f_1 \dots f_m\}$, where $f_i \in B((D \to D)_\perp)$. Define $[rep(s)]_n = \star\{[rep(f_1)]_n \dots [rep(f_m)]_n\}$.

Lemma 45 $(\forall y \in D_n)\ [[rep(s)]_n \diamond y = [App(s,y)]_n]$

Proof: Proof is by cases depending on the structure of s. Let $y \in D_n$

- s is a singleton. This splits up into two cases.
  - $s = \{\perp\}$. Result is immediate.
  - $s = \{up(f)\}$, where $f \in B(D \to D)$. Then

$$\begin{aligned}[rep(s)]_n \diamond y &= Ap_n[up(\lambda y \in D_n.[App(s,y)]_n), y]\\ &= [App(s,y)]_n\end{aligned}$$

- $s = \{f_1 \dots f_m\}$, where $f_i \in B((D \to D)_\perp)$.

$$\begin{aligned}[rep(s)]_n \diamond y &= \star\{[rep(f_1)]_n \dots [rep(f_m)]_n\} \diamond y\\ &= \star\{[rep(f_1)]_n \diamond y \dots [rep(f_m)]_n \diamond y\}\\ &= \star\{[App(f_1,y)]_n \dots [App(f_m,y)]_n\}\\ &= [\star\{App(f_1,y) \dots App(f_m,y)\}]_n\\ &= [App(\{f_1, \dots f_m\}, y)]_n \qquad \blacksquare\end{aligned}$$

Lemma 46 $s_1 \sqsubseteq s_2 \Rightarrow [rep(s_1)]_n \sqsubseteq [rep(s_2)]_n$

Proof: Let $s_1 = \{f_1 \dots f_m\}$, $s_2 = \{g_1 \dots g_n\}$.

-

$$\begin{aligned}\perp \in [rep(s_2)]_n &\Rightarrow \perp \in s_2\\ &\Rightarrow \perp \in s_1\\ &\Rightarrow \perp \in [rep(s_1)]_n\end{aligned}$$

- $[rep(s_1)]_n \neq \{\perp\} \Rightarrow [rep(s_2)]_n \neq \{\perp\}$ is proved similarly.

- Let $y \in D_n$. Then,

$$\begin{aligned}[rep(s_1)]_n \diamond y &= [App(s_1,y)]_n\\ &\sqsubseteq [App(s_2,y)]_n\\ &= [rep(s_2)]_n \diamond y \qquad \blacksquare\end{aligned}$$

As a corollary, we get that $[rep()]_n$ is well-defined on the equivalence classes of the preorder $P(D)$, and hence on $B(\overline{P}(D))$. Extend $[rep()]_n$ continuously to the whole of $\overline{P}(D)$. Continuity of the functions involved means that lemma 45 holds for the continuous extensions so defined.

Lemma 47 $j_{n+1}([rep(s)]_{n+1}) = [rep(s)]_n$

Proof: From linearity of j_n, $[rep()]_n$, suffices to prove the result for singletons i.e s is of form, $s = \{f\}$, where $f \in B((D \to D)_\perp)$. This splits up into the following two cases:

- $f = \perp$. Result follows immediately.
- $f = up(g)$, for some $g \in D \to D$.

$$\begin{aligned}
j_{n+1}([rep(s)]_{n+1}) &= j_{n+1}(\lambda y \in D_{n+1}.[App(s,y)]_{n+1}) \\
&= up(\lambda y \in D_n.j_n \circ [\lambda y \in D_{n+1}.[App(s,y)]_{n+1}] \circ i_n(y)) \\
&= up(\lambda y \in D_n.j_n([App(s,i_n(y))]_{n+1}) \\
&= up(\lambda y \in D_n.[App(s,i_n(y))]_n)\ [y \in D_n \Rightarrow i_n(y) = y] \\
&= up(\lambda y \in D_n.[App(s,y)]_n) \\
&= [rep(s)]_n \qquad \blacksquare
\end{aligned}$$

Define $rep() : \overline{P}(D) \to D$ as
$rep(s) = \bigsqcup_{n \in \omega}[rep(s)]_n$. The above lemma shows that the lub is well-defined. Furthermore, we have the following lemma.

Lemma 48 *Let $y \in D$. Then, $rep(s) \diamond y = App(s,y)$*

Proof: Let $y \in D$.

$$\begin{aligned}
rep(s) \diamond y &= \bigsqcup_{i \in \omega} App_i((rep(s))_{i+1}, y_i) \\
&= \bigsqcup_{i \in \omega}[rep(s) \diamond y_i]_i \\
&= \bigsqcup_{i \in \omega}[\bigsqcup_{n \in \omega}[rep(s)]_n \diamond y_i]_i \\
&= \bigsqcup_{i,n \in \omega}[[rep(s)]_n \diamond y_i]_i \\
&= \bigsqcup_{j \in \omega}[[rep(s)]_j \diamond y_j]_j
\end{aligned}$$

$$\begin{aligned}
&= \bigsqcup_{j\in\omega}[App(s, y_j)]_j \\
&= \bigsqcup_{n\in\omega}\bigsqcup_{i\in\omega}[App(s, y_i)]_n \\
&= \bigsqcup_{n\in\omega}[App(s, y)]_n \\
&= App(s, y) \qquad \blacksquare
\end{aligned}$$

Define $Fun : D \to \overline{P}(D)$ as follows. We first define it on $B(D)$. Note that $B(D) = \bigcup_{n\in\omega} D_n$. Let $s \in B(D)$. This definition is done by cases.

- $s = \{\bot\}$. Define $Fun(s) = \{\bot\}$.
- $s = \{up(g)\}$, for some $g \in D_n \to D_n$. Define $Fun(s) = up(x \mapsto s \diamond x)$.
- $s = \{f_1 \ldots f_m\}$. Define $Fun(s) = \{Fun(f_1), \ldots, Fun(f_m)\}$

Lemma 49 *$App(Fun(s), y) = s \diamond y$*

Proof: From linearity of *App* and *Fun* suffices to prove the result for singletons s. For singletons, result follows directly from the definition. $\blacksquare$

Lemma 50 *Fun is monotone.*

Proof: Let s_1, $s_2 \in D_n$ and $s_1 \sqsubseteq s_2$. Let $s_1 = \{f_1 \ldots f_m\}$. Let $s_2 = \{g_1 \ldots g_n\}$.

- $$\begin{aligned}
\bot \in Fun(s_2) &\Rightarrow \bot \in s_2 \\
&\Rightarrow \bot \in s_1 \\
&\Rightarrow \bot \in Fun(s_1)
\end{aligned}$$
- $Fun(s_1) \neq \{\bot\} \Rightarrow Fun(s_2) \neq \{\bot\}$ is proved similarly.
- Let $y \in D$. Then,

$$\begin{aligned}
App(Fun(s_1), y) &= s_1 \diamond y \\
&\sqsubseteq s_2 \diamond y \\
&= App(Fun(s_2), y) \qquad \blacksquare
\end{aligned}$$

Extend Fun to a continuous function in $D \to \overline{P}(D)$. From continuity of all functions involved we get $App(Fun(s), y) = s \diamond y$

Lemma 51 $(Fun \circ rep) = id_{\overline{P}(D)}$

Proof: Fun, rep are linear and continuous. So, it suffices to prove $Fun \circ rep(s) = s$, for $s = \{f\}$, where $f \in B((D \to D)_\perp)$. We have the following two cases.

- $f = \perp$. Result follows from definitions.
- $f = up(g)$, for some $g \in B(D \to D)$.

$$\begin{aligned} Fun(rep(s)) &= \{up(x \mapsto rep(s) \diamond x)\} \\ &= \{up(x \mapsto App(s, x))\} \\ &= \{up(x \mapsto g(x))\} \\ &= s \quad \blacksquare \end{aligned}$$

Lemma 52 $(rep \circ Fun) = id_D$

Proof: Since Fun, rep are linear, continuous, it suffices to check $(rep \circ Fun)(s) = s$, for $s = \{f\}$, $s \in D_{n+1}$, for some n. We have the following two cases.

- $f = \perp$. Result follows from definitions.
- $f = up(g)$, for some $g \in D_n \to D_n)$.

$$\begin{aligned} rep(Fun(s)) &= rep(up(x \mapsto s \diamond x)) \\ &= \bigsqcup_{n \in \omega}[[rep(up(x \mapsto s \diamond x))]_n] \\ &= \bigsqcup_{n \in \omega}[up(\lambda y \in D_n.[App(up(x \mapsto s \diamond x), y)]_n)] \\ &= \bigsqcup_{n \in \omega}[up(\lambda y \in D_n.[s \diamond y]_n)] \\ &= \bigsqcup_{n \in \omega}[up(\lambda y \in D_n.[s_{n+1} \diamond y])] \\ &= \bigsqcup_{n \in \omega} s_{n+1} \\ &= s \quad \blacksquare \end{aligned}$$

Lemma 53 *Let* $s_1, s_2 \in D_n$, $1 \leq n$. *Then, the following are equivalent:*

- $(s_1)_1 \sqsubseteq (s_2)_1$
- $[\perp \in s_2 \Rightarrow \perp \in s_1] \wedge [s_1 \neq \{\perp\} \Rightarrow s_2 \neq \{\perp\}]$

Proof: By induction on n, where $1 \leq n$. ∎

Lemma 54 *(Conditional Strong extensionality) Let $d, e \in D$. Then*
$d \sqsubseteq e \Leftrightarrow d_1 \sqsubseteq e_1 \wedge \ (\forall x \in D)\ [d \diamond x \sqsubseteq e \diamond x]$

Proof: Forward implication is immediate. For the reverse implication, note that $(\forall x \in D_n) d_{n+1} \diamond x = [d \diamond x]_n \sqsubseteq [e \diamond x]_n = e_{n+1} \diamond x$. This along with the previous lemma shows that $d_n \sqsubseteq e_n$. ∎

3.4.3 Product structure on D

The | constructor is modeled by a continuous function $\times : D \times D \rightarrow D$. In this section we use $[]_1$ as shorthand for the projection map onto D_1. The following lemma is used implicitly in the following proofs.

Lemma 55 *Let $f \in D_s$. Then,*

- $\perp \notin f \Leftrightarrow \{\lambda x.\perp_0\} \sqsubseteq [f]_1$
- $\perp = f \Leftrightarrow \perp = [f]_1$

Proof: Induction on s. ∎

Let D_s be the iterates in the solution of the recursive domain equation $D \simeq \overline{P}(D)$. Define a family of functions $\times_{(s,t)} : D_s \times D_t \rightarrow D_{(s+t)}$, by induction on $s+t$ as follows. Let $f \in D_s$, $g \in D_t$.

- $(s+t=0)$. $f \times_{(0,0)} g = \perp_{(0,0)}$
- $(s+t \neq 0)$. Assume f, g are singleton sets. Then, define by cases on $[f]_1$, $[g]_1$ as follows.
  - $\{\lambda x.\perp_0\} = [f]_1$, $\perp_0 = [g]_1$. Then,
    $f \times_{(s,t)} g = \perp_{(s+t)} \star up[x \in D_{s+t-1} \mapsto (f \diamond x) \times_{(s-1,t)} \perp_t]$
  - $\{\lambda x.\perp_0\} = [g]_1$, $\perp_0 = [f]_1$. Then,
    $f \times_{(s,t)} g = \perp_{(s+t)} \star [up[x \in D_{s+t-1} \mapsto \perp_s \times_{(s,t-1)} (g \diamond x)]$
  - $\{\lambda x.\perp_0\} = [f]_1$, $\{\lambda x.\perp_0\} = [g]_1$,. Then,
    $f \times_{(s,t)} g = up[x \in D_{s+t-1} \mapsto ([f \times_{(s,t-1)} (g \diamond x)] \star [(f \diamond x) \times_{(s-1,t)} g])$
  - $\perp_0 = [f]_1$, $\perp_0 = [g]_1$. Then,
    $f \times_{(s,t)} g = \perp_{(s+t)}$.

- $(s+t \neq 0)$. $f = \{[f]_1 \dots f_m\}$, $g = \{g_1 \dots g_n\}$. Then,
  $f \times_{(s,t)} g = \star[f_i \times_{(s,t)} g_j | 1 \leq i \leq m,\ 1 \leq j \leq n]$

We first show that $\times_{(s,t)}$ is well-defined and monotone in both its arguments.

Lemma 56 *$(\forall s,\ t \in \omega)$, the following holds. Let $f \in D_s$, $g \in D_t$. Then,*

- $[f \times_{(s,t)} g] \in D_{(s+t)}$
- *Let $f' \in D_s$, $g' \in D_t$, $f \sqsubseteq f'$, $g \sqsubseteq g'$. Then, $f \times_{(s,t)} g \sqsubseteq f' \times_{(s,t)} g'$.*

Proof: The proof of both parts is by induction on $s+t$. The base case $s+t=0$, is checked easily. For the induction step, assume result for $s+t \leq n$. Consider $s+t = n+1$. Let $f = \{f_1 \dots f_m\}$, $g = \{g_1 \dots g_n\}$, such that $f \in D_s$, $g \in D_t$.

1. (Proving that $f \times_{(s,t)} g \in D_{(s+t)}$).
   Thus we need to prove the monotonicity of $f \times_{(s,t)} g$, as a function from $D_{(s+t-1)}$ to $D_{(s+t-1)}$.

   - $(\{\lambda x.\bot_0\} \sqsubseteq [f]_1,\ \{\lambda x.\bot_0\} \sqsubseteq [g]_1)$. In the following let $1 \leq i \leq m$, $1 \leq j \leq n$. Then,

$$
\begin{aligned}
(f \times_{(s,t)} g) \diamond x &= \star[f_i \times_{(s,t)} g_j | i,j] \diamond x \\
&= \star[(f_i \times_{(s,t)} g_j) \diamond x | i,j] \\
&= \star[((f_i \diamond x) \times_{(s-1,t)} g_j) \star (f_i \times_{(s,t-1)} (g_j \diamond x)) | i,j]
\end{aligned}
$$

     The result follows from induction hypothesis and the monotonicity of $\star$.

   - $(\{\lambda x.\bot_0\} \sqsubseteq [f]_1,\ \{\lambda x.\bot_0\} \not\sqsubseteq [g]_1)$. Let $g_1 = \bot$. Then, In the following let $1 \leq i \leq m$, $2 \leq j \leq n$. Then,

$$
\begin{aligned}
(f \times_{(s,t)} g) \diamond x &= \star[f_i \times_{(s,t)} g_j | i,j] \diamond x \\
&\quad \star \bot \star\ [(\star[f_i \times_{(s,t)} \bot]) | i] \diamond x] \\
&= [((f_i \diamond x) \times_{(s-1,t)} g_j) \star (f_i \times_{(s,t-1)} (g_j \diamond x)) | i,j] \\
&\quad \star \bot \star\ [((f_i \diamond x) \times_{(s-1,t)} \bot) | i]
\end{aligned}
$$

     The result follows from induction hypothesis and the monotonicity of $\star$.

   - $(\{\lambda x.\bot_0\} \not\sqsubseteq [f]_1,\ \{\lambda x.\bot_0\} \sqsubseteq [g]_1)$. Proof is similar to the preceding case.

- $(\{\lambda x.\perp_0\} \not\sqsubseteq [f]_1,\ \{\lambda x.\perp_0\} \not\sqsubseteq [g]_1)$. Proof is similar to the previous cases.

2. (Proving monotonicity of $(\times_{(s,t)})$
We prove the monotonicity of $\times_{(s,t)}$ in its left argument. Proof of monotonicity of $\times_{(s,t)}$ in its right argument is similar, and is omitted. Let $h = \{h_1 \ldots h_l\}$, $h \in D_s$, $f \sqsubseteq h$. Then,

- Let $\perp \in h \times_{(s,t)} g$. Then,

$$\begin{aligned} \perp \in h \times_{(s,t)} g &\Rightarrow \perp \in h \vee \perp \in g \\ &\Rightarrow \perp \in f \vee \perp \in g \\ &\Rightarrow \perp \in f \times_{(s,t)} g \end{aligned}$$

- $\perp \neq f \times_{(s,t)} g \Rightarrow \perp \neq h \times_{(s,t)} g$ is proved similarly.

So, proof is complete if we check
$(\forall x \in D_{s+t})\ [(f \times_{(s,t)} g) \diamond x \sqsubseteq (h \times_{(s,t)} g) \diamond x]$. This splits up into the following cases.

- $(\{\lambda x.\perp_0\} \sqsubseteq [f]_1,\ \{\lambda x.\perp_0\} \sqsubseteq [g]_1)$. So, $\{\lambda x.\perp_0\} \sqsubseteq [h]_1$, as $f \sqsubseteq h$. In the following let $1 \leq i \leq m,\ 1 \leq j \leq n$. Then,

$$\begin{aligned} (f \times_{(s,t)} g) \diamond x &= (\star[f_i \times_{(s,t)} g_j | i,j]) \diamond x \\ &= \star[(f_i \times_{(s,t)} g_j) \diamond x | i,j] \\ &= \star[((f_i \diamond x) \times_{(s-1,t)} g_j) \star (f_i \times_{(s,t-1)} (g_j \diamond x)) | i,j] \\ &= [(f \diamond x) \times_{(s-1,t)} g_j] \star [f \times_{(s-1,t)} (g_j \diamond x | j] \end{aligned}$$

  Similarly,

$$(h \times_{(s,t)} g) \diamond x = [(h \diamond x) \times_{(s-1,t)} g_j] \star [h \times_{(s-1,t)} (g_j \diamond x) | j]$$

  The result now follows from induction hypothesis, monotonicity of $\star$ and noting that $f \diamond x \sqsubseteq h \diamond x$

- $(\{\lambda x.\perp_0\} \not\sqsubseteq [f]_1,\ \{\lambda x.\perp_0\} \sqsubseteq [g]_1,\ \{\lambda x.\perp_0\} \sqsubseteq [h]_1)$. Then, $\perp \in f$. Without loss of generality, assume that $f_1 = \perp$. In the following let $1 \leq i \leq m, 2 \leq$

$i' \leq n, 1 \leq j \leq n$.

$$\begin{aligned}
(f \times_{(s,t)} g) \diamond x &= (\star[f_i \times_{(s,t)} g_j | i, j]) \diamond x \\
&= \star[(f_i \times_{(s,t)} g_j) \diamond x | i, j] \\
&= \star[((f_i \diamond x) \times_{(s-1,t)} g_j) \star (f_i \times_{(s,t-1)} (g_j \diamond x)) | i', j] \\
&\quad \star\perp \ \star [\perp \times_{(s,t-1)} (g_j \diamond x)) | j]
\end{aligned}$$

Furthermore, if $1 \leq k \leq l$, $1 \leq j \leq n$,

$$\begin{aligned}
(h \times_{(s,t)} g) \diamond x &= \star[h_k \times_{(s,t)} g_j | k, j] \diamond x \\
&= \star[(h_k \times_{(s,t)} g_j) \diamond x | k, j] \\
&= \star[(h_k \times_{(s,t)} g_j) \diamond x | k, j] \\
&= \star[((h_k \diamond x) \times_{(s-1,t)} g_j) \star (h_k \times_{(s,t-1)} (g_j \diamond x)) | k, j] \\
&= \star[((h \diamond x) \times_{(s-1,t)} g_j) \star (h \times_{(s,t-1)} (g_j \diamond x)) | j]
\end{aligned}$$

$[(f_i \times_{(s,t-1)} (g_j \diamond x)) | 2 \leq i \leq m] \star [\perp \times_{(s,t-1)} (g_j \diamond x)] = f \times_{(s,t-1)} (g_j \diamond x)$. From induction hypothesis, $f \times_{(s,t-1)} (g_j \diamond x) \sqsubseteq h \times_{(s,t-1)} (g_j \diamond x)$. Also, if $2 \leq i \leq m$,

$$\begin{aligned}
\perp \star [(f_i \diamond x) \times_{(s-1,t)} g_j | i] &\sqsubseteq (\perp \times_{(s-1,t)} g_j) \star [(f_i \diamond x) \times_{(s-1,t)} g_j | i] \\
&= (f \diamond x) \times_{(s-1,t)} g_j \\
&\sqsubseteq (h \diamond x) \times_{(s-1,t)} g_j
\end{aligned}$$

Hence, the result.

- $(\{\lambda x.\perp_0\} \not\sqsubseteq [f]_1,\ \{\lambda x.\perp_0\} \sqsubseteq [g]_1,\ \{\lambda x.\perp_0\} \not\sqsubseteq [h]_1)$. As in previous case, if $2 \leq i \leq m$, $1 \leq j \leq n$,

$$\begin{aligned}
(f \times_{(s,t)} g) \diamond x &= \star[((f_i \diamond x) \times_{(s-1,t)} g_j) \star (f_i \times_{(s,t-1)} (g_j \diamond x)) | i, j] \\
&\quad \star\perp \ \star [\perp \times_{(s,t-1)} (g_j \diamond x)) | j]
\end{aligned}$$

Also by a similar argument, if $2 \leq k \leq l$, $1 \leq j \leq n$,

$$\begin{aligned}
(h \times_{(s,t)} g) \diamond x &= \star[((h_k \diamond x) \times_{(s-1,t)} g_j) \star (h_k \times_{(s,t-1)} (g_j \diamond x)) | k, j] \\
&\quad \star\perp \ \star \ [\perp \times_{(s,t-1)} (g_j \diamond x)) | j]
\end{aligned}$$

As in previous case, if $2 \leq i \leq m, 2 \leq k \leq l$, we have

$$\begin{aligned}
[(f_i \times_{(s,t-1)} (g_j \diamond x))|i] &\star [\perp \times_{(s,t-1)} (g_j \diamond x)] \\
&= f \times_{(s,t-1)} (g_j \diamond x) \\
&\sqsubseteq h \times_{(s,t-1)} (g_j \diamond x) \\
&= [h_k \times_{(s,t-1)} (g_j \diamond x)|k] \star [\perp \times_{(s,t-1)} (g_j \diamond x)]
\end{aligned}$$

Also, if $2 \leq i \leq m, 2 \leq k \leq l$,

$$\begin{aligned}
\perp \star [(f_i \diamond x) \times_{(s-1,t)} g_j|i] &\sqsubseteq (\perp \times_{(s-1,t)} g_j) \star [(f_i \diamond x) \times_{(s-1,t)} g_j|i] \\
&= (f \diamond x) \times_{(s-1,t)} g_j \\
&\sqsubseteq [((h_k \diamond x) \times_{(s-1,t)} g_j)|k]
\end{aligned}$$

The last step follows because

$$\begin{aligned}
f \diamond x &\sqsubseteq h \diamond x \\
&\sqsubseteq [h_k \diamond x|\ 2 \leq k \leq l]
\end{aligned}$$

So, from idempotence and monotonicity of $\star$, and induction hypothesis, if $2 \leq i \leq m, 2 \leq k \leq l$,

$$\perp \star [(f_i \diamond x) \times_{(s-1,t)} g_j|i] \sqsubseteq [((h_k \diamond x) \times_{(s-1,t)} g_j)|k] \star \perp$$

Hence, the result.

The proofs of the other cases are similar and are omitted. ∎

The following lemma shows that the subscripts can be dropped from $\times_{(s,t)}$.

Lemma 57 *(Well-definedness of $\times_{(s,t)}$)*
Let $f \in D_s$, $g \in D_t$. Then $i_s(f) \times_{(s+1,t)} g = i_{(s+t)}(f \times_{(s,t)} g)$.

Proof: We prove that $i_s(f) \times_{(s+1,t)} g = i_{(s+t)}[f \times_{(s,t)} g]$, from which result follows. Proof is by induction on $s + t$. The base case $s + t = 0$ follows immediately. Assume result for $s + t \leq b$. Consider the case $s + t = n + 1$. From the linearity of i_s, $i_{(s+t)}$ and the bilinearity of $\times_{(s,t)}$, $\times_{(s+1,t)}$, it suffices to prove the result for singletons f, g. Thus we have the following cases.

- $\{\lambda x.\perp\} = [f]_1$, $\{\lambda x.\perp\} = [g]_1$. Then,
  $i_{(s+t)}(f \times_{(s,t)} g) = i_{(s+t)}(up(x \mapsto [((f \diamond x) \times_{(s-1,t)} g) \star (f \times_{(s-1,t)} (g \diamond x))]$. The first two conditions of definition 42 are verified easily. So, we just need to check that
  $(\forall x \in D_{(s+t)}[(i_{(s+t)}(f \times_{(s,t)} g)) \diamond x = (i_s(f) \times_{(s+1,t)} g) \diamond x]$.
  Let $x \in D_{(s+t)}$. Let $y = j_{(s+t-1)}(x)$. Then, $(i_{(s+t)}(f \times_{(s,t)} g)) \diamond x =$

$$= i_{(s+t-1)}[(f \diamond y) \times_{(s-1,t)} g) \star (f \times_{(s,t-1)} (g \diamond y)]$$
$$= i_{s+t-1}[((f \diamond y) \times_{(s-1,t)} g) \star \; i_{(s+t-1)}(f \times_{(s,t-1)} (g \diamond y))]$$
$$= (i_{s-1}(f \diamond y) \times_{(s-1,t)} g) \star \; (i_s(f) \times_{(s,t-1)} (g \diamond y))$$

  The last step follows from the induction hypothesis. Since $f \in D_s$,

$$\begin{aligned} i_{s-1}(f \diamond y) &= i_{s-1}(f \diamond y_{(s-1)}) \quad [s \leq s+t-1 \Rightarrow y_{(s-1)} = x_{(s-1)}] \\ &= i_{s-1}(f \diamond x_{(s-1)}) \\ &= i_{s-1}(f \diamond x) \\ &= f \diamond x \qquad [i_s(f) = f] \\ &= i_s(f) \diamond x \end{aligned}$$

  Hence the result.

- $\{\lambda x.\perp\} = [g]_1$, $\perp = [f]_1$. Then,
  $f \times_{(s,t)} g = \perp \; \star \; up(x \mapsto \; [\perp \times_{(s,t-1)}])(g \diamond x)$. So,
  $i_{(s+t)}(f \times_{(s,t)} g) = i_{(s+t)}(\perp) \star \; i_{(s+t)}(up(x \mapsto \; (\perp \times_{(s,t-1)} (g \diamond x))))$.
  The first two conditions in the definition 42 are checked easily.
  Let $x \in D_{(s+t)}$, $y = j_{(s+t-1)}(x)$. Then,

$$\begin{aligned} i_{(s+t)}(\perp) \star \; (i_{(s+t)(\perp \times_{(s,t)} g))}x &= \perp \; \star \; i_{(s+t-1)}[\perp \; \star (\perp \times_{(s,t-1)} (g \diamond y))] \\ &= \perp \; \star i_{s+t-1}[\perp] \star \; i_{s+t-1}[\perp \times_{(s+1,t-1)} g \diamond y] \\ &= \perp \star \; (i_s(\perp) \times_{(s+1,t-1)} (g \diamond y)) \\ &= \perp \star \; (i_s(\perp) \times_{(s+1,t-1)} (g \diamond x)) \\ &= ((i_s(\perp)) \times_{(s+1,t)} g) \diamond x \end{aligned}$$

  $x \in D_{(s+t)} \Rightarrow [g \diamond x = g \diamond x_{(t-1)}]$, and $x_{(t-1)} = y_{(t-1)}$.

- $\{\lambda x.\bot\} = [f]_1,\ \bot = [g]_1$. Then,

$$f \times_{(s,t)} g = \bot \star up(x \in D_{(s+t-1)} \mapsto [f \diamond x \times_{(s-1,t)} \bot])$$

So,

$$i_{(s+t)}(f \times_{(s,t)} g) = \bot \star i_{(s+t)}[up(x \in D_{(s+t-1)} \mapsto [f \diamond x \times_{(s-1,t)} \bot])]$$

The first two conditions in the definition 42 are checked easily.

Let $x \in D_{(s+t)}$, $y = j_{(s+t-1)}(x)$. Then,

$$\begin{aligned}
(i_{(s+t)}(f \times_{(s,t)} g)) \diamond x &= \bot \star i_{(s+t-1)}[\bot \star (f \diamond x) \times_{(s-1,t)} \bot] \\
&= \bot \star i_{(s+t-1)}[\bot] \star i_{(s+t-1)}[(f \diamond y) \times_{(s,t-1)} \bot] \\
&= \bot \star [(i_{(s-1)}(f \diamond y)) \times_{s,t-1} \bot]\ (f \diamond y = f_s \diamond y_{s-1}) \\
&= \bot \star [(f \diamond y)) \times_{(s,t-1)} \bot] \quad (f = i_s(f)) \\
&= \bot \star [(i_s(f) \diamond y) \times_{(s,t-1)} \bot] \ (i_s(f) \diamond x = i_s(f) \diamond y) \\
&= \bot \star [(i_s(f) \diamond x) \times_{(s,t-1)} \bot] \\
&= ((i_s(f)) \times_{(s+1,t)} \bot) \diamond x \qquad \blacksquare
\end{aligned}$$

So, we can drop the subscripts on the $\times$. The above lemmas ensure that $\times : B(D) \times B(D) \to B(D)$ is well-defined and monotone. Extend $\times$ to a continuous function $\times : D \times D \to D$. The following lemma delineates some algebraic properties that describes the interaction between $\times$, $\star$ and $\times$, $\diamond$. The proofs are done for finite elements of D. The results for arbitrary elements of D follow from the continuity of all functions involved.

Lemma 58 *Let d, $,e$, f, $x \in B(D)$. Then*

1. $d \times (e \star f) = (d \times f) \star (e \times f)$
2. $(d \times \bot) \star \bot = (d \times \bot)$
3. $\bot \times \bot = \bot$
4. $d \times e = e \times d$
5. $[(d \uplus \bot \neq d) \wedge (e \uplus \bot \neq e)] \Rightarrow$
   $(d \times e) \diamond f = ((d \diamond f) \times e) \uplus (d \times (e \diamond f))$

6. $(d \uplus \perp \neq d) \Rightarrow$

   $(d \times e) \diamond f = (d \times e) \diamond f \uplus d \diamond f$

7. $d \times (e \times f) = (d \times e) \times f)$

Proof: The proofs of 1, 2, 3, 4, 5, 6 are immediate from the definitions of the indexed version of $\times$. 7 is proved below. The proof uses the indexed version of the definition of $\times$. We prove

$(f \times_{(s,t)} g) \times_{(s+t,u)} h = f \times_{(s,t+u)} (g \times_{(t,u)} h$. The proof proceeds by induction on $s+t+u$. The base case is immediate. For the induction step, from the bilinearity of $\times_{(s,t)}$, $\times_{(s,t+u)}$, it suffices to show the result for singletons f, g, h. The proof proceeds by cases on $[f]_1$, $[g]_1$, $[h]_1$.

1. $[f]_1 = \perp$, $[g]_1 = \perp$, $[h]_1 = \perp$. Result is immediate.

2. $[f]_1 \neq \perp$, $[g]_1 \neq \perp$, $[h]_1 \neq \perp$. The first two clauses in definition 42 are verified easily. So, proof is complete if

   $(\forall x \in D_{(s+t+u-1)}) \; [((f \times_{(s,t)} g) \times_{(s+t,u)} h) \diamond x = (f \times_{(s,t+u)} (g \times_{(t,u)} h)) \diamond x]$. From definitions,

   $((f \times_{(s,t)} g) \times_{(s+t,u)} h) \diamond x$

   $= [((f \times_{(s,t)} g) \diamond x) \times_{(s+t-1,u)} h] \star \; [(f \times_{(s,t)} g)] \times_{(s+t,u-1)} (h \diamond x)]$

   $= [(f \diamond x \times_{(s-1,t)} g) \star (f \times_{(s,t-1)} g \diamond x) \times_{(s+t-1,u)} h] \star \; [(f \times_{(s,t)} g)] \times_{(s+t,u-1)} (h \diamond x)]$

   $= ((f \diamond x \times_{(s-1,t)} g) \times_{(s+t-1,u)} h) \star ((f \times_{(s,t)} g \diamond x) \times_{(s+t-1,u)} h) \star \; ((f \times_{(s,t)} g) \times_{(s+t,u-1)}$ $(g \diamond x))$ Indn. Hyp

   $= (f \diamond x \times_{(s-1,t+u)} (g \times_{(t,u)} h)) \star (f \times_{(s,t+u-1)} (g \diamond x) \times_{(t-1,u)} h) \star \; (f \times_{(s,t+u-1)}$ $(g \times_{(t,u-1)} h \diamond x))$

   $= (f \diamond x \times_{(s-1,t+u)} (g \times_{(t,u)} h)) \star (f \times_{(s,t+u-1)} ((g \times_{(t,u)} h) \diamond x))$

3. $[f]_1 = \perp$, $[g]_1 = \perp$, $[h]_1 \neq \perp$. From definitions, $(f \times_{(s,t)} g) \times_{(s+t,u)} h =$, $\perp \star \; up(x \in D_{(s+t+u-1)} \mapsto [\perp \times_{(s+t,u-1)} (h \diamond x)])$. $f \times_{(s,t+u)} (g \times_{(t,u)} h) =$ $= \perp \star \; \perp \times_{(s,t+u)} (up(x \in D_{(t+u-1)} \mapsto [\perp \times_{(t,u-1)} (h \diamond x)]))$.

   The first two clauses in definition 42 are verified easily. We prove that $(\forall x \in D_{(s+t+u-1)}) \; [((f \times_{(s,t)} g) \times_{(s+t,u)} h) \diamond x = (f \times_{(s,t+u)} (g \times_{(t,u)} h)) \diamond x]$. From definitions,

   $$((f \times_{(s,t)} g) \times_{(s+t,u)} h) \diamond x \;\; = \;\; \perp \star \; (\perp \times_{(s+t,u-1)} (h \diamond x))$$

Also,

$$\begin{aligned}(f \times_{(s,t+u)} (g \times_{(t,u)} h)) \diamond x &= \perp\star\ (\perp \times_{(s,t+u-1)} (\perp \times_{(t,u-1)} (h \diamond x))) \text{ Indn.}\\ &= \perp\star\ ((\perp \times_{(s,t)} \perp) \times_{(s+t,u-1)} (h \diamond x))\\ &= \perp\star\ ((\perp \times_{(s+t,u-1)} (h \diamond x))\end{aligned}$$

4. $[f]_1 \neq \perp$, $[g]_1 = \perp$, $[h]_1 = \perp$. As in the preceding case.

5. $[f]_1 = \perp$, $[g]_1 \neq \perp$, $[h]_1 = \perp$. Follows from Part 4 of the lemma.

6. $[f]_1 = \perp$, $[g]_1 \neq \perp$, $[h]_1 \neq \perp$. From definitions, $(f \times_{(s,t)} g) \times_{(s+t,u)} h =$

$$\begin{aligned}&= [\perp\star\ up(x \in D_{(s+t-1)} \mapsto (\perp \times_{(s,t-1)} g \diamond x))] \times_{(s+t,u)} h\\ &= (\perp \times_{(s+t,u)} h) \star\ (up(x \in D_{(s+t-1)} \mapsto (\perp \times_{(s,t-1)} g \diamond x)) \times_{(s+t,u)} h)\\ &= \perp\star\ up(x \in D_{(s+t+u-1)} \perp \times_{(s+t,u-1)} (h \diamond x))\\ &\star\ (up(x \in D_{(s+t)} \mapsto (\perp \times_{(s,t-1)} g \diamond x))) \times_{(s+t,u)} h)\end{aligned}$$

So, we have, $((f \times_{(s,t)} g) \times_{(s+t,u)} h) \diamond x =$

$$\begin{aligned}&= \perp\star\ \perp \times_{(s+t,u-1)} (h \diamond x) \star\ ((\perp \times_{(s,t-1)} g \diamond x) \times_{(s+t-1,u)} h)\\ &\star\ (up(x \in D_{(s+t-1)} \mapsto (\perp \times_{(s,t-1)} g \diamond x))) \times_{(s+t,u-1)} h \diamond x\end{aligned}$$

From definitions,

$f \times_{(s,t+u)} (g \times_{(t,u)} h)$

$= f \times_{(s,t+u)} (up(x \in D_{(t+u-1)} \mapsto [(g \times_{(t,u-1)} h \diamond x) \star (g \diamond x \times_{(t-1,u)} h)]))$

$= \perp\ \star\ (up(x \in D_{(s+t+u-1)} \mapsto [(\perp \times_{(s,t+u-1)} (g \times_{(t,u-1)} h \diamond x)) \star (\perp \times_{(s,t+u-1)}$ $(g \diamond x \times_{(t-1,u)} h))]))$

So, we have

$(f \times_{(s,t+u)} (g \times_{(t,u)} h)) \diamond x$

$= \perp\ \star\ (\perp \times_{(s,t+u-1)} (g \times_{(t,u-1)} h \diamond x))$

$\star (\perp \times_{(s,t+u-1)} (g \diamond x \times_{(t-1,u)} h))$.

The first two clauses in definition 42 are verified easily. We prove

$$(\forall x \in D_{(s+t+u-1)}) [((f \times_{(s,t)} g) \times_{(s+t,u)} h) \diamond x = (f \times_{(s,t+u)} (g \times_{(t,u)} h)) \diamond x]$$

From induction hypothesis, $\perp \times_{(s,t+u-1)} (g \diamond x \times_{(t-1,u)} h)$
$= \perp \times_{(s,t+u-1)} (g \diamond x \times_{(t-1,u)} h)$.
Also,

$$\begin{aligned}
\perp \quad \times_{(s+t,u-1)} \quad & (h \diamond x) \star up(x \in D_{(s+t-1)} \mapsto \perp \times_{(s,t-1)} g \diamond x) \times_{(s+t,u-1)} h \diamond x \\
= \quad & (\perp \star (up(x \in D_{(s+t-1)} \mapsto (\perp \times_{(s,t-1)} g \diamond x)))) \times_{(s+t,u-1)} h \diamond x \\
= \quad & (\perp \times_{(s,t)} g) \times_{(s+t,u-1)} h \diamond x \text{ Indn. Hyp} \\
= \quad & \perp \times_{(s,t+u-1)} (g \times_{(t,u-1)} h \diamond x)
\end{aligned}$$

Hence, the result.

7. $[f]_1 \neq \perp$, $[g]_1 = \perp$, $[h]_1 = \perp$. As in preceding case.

8. $[f]_1 \neq \perp$, $[g]_1 = \perp$, $[h]_1 \neq \perp$. From definitions, $(f \times_{(s,t)} g) \times_{(s+t,u)} h =$

$$\begin{aligned}
= \quad & (\perp \star \ up(x \in D_{s+t-1} \mapsto [(f \diamond x) \times_{(s-1,t)} \perp])) \times_{(s+t,u)} h \\
= \quad & (\perp \times_{(s+t,u)} h) \star \ (up(x \in D_{s+t-1} \mapsto [(f \diamond x) \times_{(s-1,t)} \perp]) \times_{(s+t,u)} h) \\
= \quad & (\perp \star up(x \in D_{(s+t+u-1)} \mapsto \perp \times_{(s+t,u-1)} h \diamond x)) \\
& \star (up(x \in D_{s+t-1} \mapsto [(f \diamond x) \times_{(s-1,t)} \perp]) \times_{(s+t,u)} h)
\end{aligned}$$

Let $x \in D_{(s+t+u-1)}$. We have $((f \times_{(s,t)} g) \times_{(s+t,u)} h) \diamond x$
$= \perp \star (\perp \times_{(s+t,u-1)} h \diamond x)$
$\star \ (((f \diamond x) \times_{(s-1,t)} \perp) \times_{(s+t-1,u)} h)$
$\star \ (up(x \in D_{s+t-1} \mapsto [(f \diamond x) \times_{(s-1,t)} \perp]) \times_{(s+t,u-1)} h \diamond x)$.
From definitions, $f \times_{(s,t+u)} (g \times_{(t,u)} h)$
$= f \times_{(s,t+u)} (\perp \star \ up(x \in D_{(t+u-1)} \mapsto \perp \times_{(t,u-1)} (h \diamond x)))$
$= (f \times_{(s,t+u)} \perp) \star \ (f \times_{(s,t+u)} up(x \in D_{(t+u-1)} \mapsto \perp \times_{(t,u-1)} (h \diamond x)))$
$= \perp \star \ up(x \in D_{(s+t+u-1)} \mapsto (f \diamond x \times_{(s-1,t+u)} \perp)) \star \ (f \times_{(s,t+u)} up(x \in D_{(t+u-1)} \mapsto$
$\perp \times_{(t,u-1)} (h \diamond x)))$.
Let $x \in D_{(s+t+u-1)}$. We have
$(f \times_{(s,t+u)} (g \times_{(t,u)} h)) \diamond x$
$= \perp \star (f \diamond x \times_{(s-1,t+u)} \perp)) \star (f \diamond x \times_{(s-1,t+u)} up(x \in D_{(t+u-1)} \mapsto \perp \times_{(t,u-1)} (h \diamond x)))$
$\star \ (f \times_{(s,t+u-1)} (\perp \times_{(t,u-1)} (h \diamond x)))$

Also,

$$\begin{array}{rl}(\perp \times_{(s+t,u-1)} & h \diamond x) \star up(x \in D_{s+t-1} \mapsto (f \diamond x) \times_{(s-1,t)} \perp) \times_{(s+t,u-1)} h \diamond x \\ = & \perp \star \; (up(x \in D_{s+t-1} \mapsto [(f \diamond x) \times_{(s-1,t)} \perp]) \times_{(s+t,u-1)} h \diamond x) \\ = & (f \times_{(s-1,t)} \perp) \times_{(s+t,u-1)} h \diamond x)\end{array}$$

Also, by induction hypothesis,

$$\begin{array}{rcl}(f \diamond x \times_{(s-1,t)} \perp) \times_{(s+t-1,u)} h & = & (f \diamond x) \times_{(s-1,t+u)} (\perp \times_{(t,u)} h) \\ & = & (f \diamond x) \times_{(s-1,t+u)} (\perp \star u)\end{array}$$

where $u = up(x \in D_{(t+u-1)} \mapsto \perp \times_{(t,u-1)} (h \diamond x))$. Result follows from the linearity of $\times_{(s-1,t+u)}$ in its right argument. ∎

3.5 Adequacy

In this section, we describe the relationship between $\perp$ and non-termination in the calculus. This is, of course, crucial if our mathematical model is to say anything interesting about computation. Our model has the following adequacy properties. Let M be a closed term. Then,

1. $\neg(M\Downarrow^{may}) \Rightarrow [\![M]\!]\,\rho = \perp$

2. $\neg(M\Downarrow^{must}) \Rightarrow [\![M]\!]\,\rho \uplus \perp = [\![M]\!]\,\rho$

3. $[\![M]\!]\,\rho \sqsubseteq [\![N]\!]\,\rho \Rightarrow M \preceq N$ for closed terms.

The proof superficially resembles the proof of adequacy in the setting of the lazy lambda calculus [4,50]. The details, however, are rather more intricate than that situation as we have to deal with many possible reduction sequences; with indeterminacy in the calculus one cannot have a deterministic evaluation strategy. The full proof is described in the succeeding sections. The overview of the proof is as follows.

We first introduce a labelled calculus and show that it is strongly normalizing. We then consider a reduction strategy $\rightarrow_\omega$. We show that any $\rightarrow_\omega$ reduction in the labelled calculus can be mimicked in the γ-calculus. We then define a semantics for the labelled calculus in terms of approximable models [50,68] equipped with extra structure to handle indeterminacy and concurrency and show that the meaning of a completely labelled term is less than the "union" of the meanings of all terms derived from one step $\rightarrow_\omega$ reductions. Because the fully labelled calculus is strongly normalizing and reduction is finitely-branching, we can classify all the "normal forms" that might exist after a fully labelled term is reduced. We can also show that the meaning of a term in the γ-calculus is given by the least upper bound of the meanings of the completely labelled terms derived from it. If we have a term M that never terminates, i.e. $\neg(M\Downarrow^{may})$, we can inspect all the terms that arise from reducing all its completely labelled versions and show that they all denote $\perp$. Thus the original term itself must have meaning bottom.

A similar but slightly more subtle argument is used for the "must converge" case. Suppose that we have a term, M, satisfying $\neg(M\Downarrow^{must})$. Reductions in the γ-calculus cannot be mimicked completely in the labelled version. However, if we examine a divergent reduction sequence of M and attempt to mimic it in the labelled calculus, we reach a point where the head redex has label 0. At this point we know that the meaning of the original term must "contain" $\perp$.

Labelled calculus

The terms of the labelled calculus with bottom, denoted $BC^\omega\perp$ is defined by the following grammar

Terms ::= $x \;||\; \lambda\langle x_1 \ldots x_k\rangle.M \;||\; MN \;||\; M|N \;||M^n$

where $n \in \omega$. Following the notation used in the study of the lazy lambda calculus [50], we have

- $|M|$ is notation for the term got by erasing the labels of M.
- *subterm*(M) denotes the set of all subterms of M
- $Seq^\star$ denotes the set of all non-empty finite sequences of ω.
- A labelled term M is formalized as a pair (M, I_M) where I_M : *subterm*$(M) \rightarrow Seq^\star \bigcup\{\infty\}$, maps a subterm N of M to the non-empty sequence of (nested)labels of N. If N has no labels then $I_M(N) = \infty$.

- The set of completely labelled labelled terms CL is defined by structural induction as follows.
  - $x^n \in CL$
  - $M \in CL,\ N \in CL \Rightarrow$
    $M^n,\ (MN)^n, (\lambda\langle x_1, \ldots x_k\rangle.M)^n, M|N \in CL$

  Define
  $CL(M) = \{I_M | (M, I_M) \text{ is completely labelled } \}$

Definition 43 *The syntactic equality $\equiv$ is the congruence (with respect to substitution) that is generated by the following equation:*

$$p|(q|r) \equiv\ (p|q)|r$$

Let $[x \mapsto N]M$ be notation for the usual notion of substitution. The reduction relation is presented as a transition system. (The presentation here is semi-formal, but is formalized clearly in the appendix where strong normalization is proved).

- $(M^m)^n \rightarrow_l M^{min(m,n)}$
- $(\lambda\langle x_1 \ldots x_k\rangle.M)^{n+1} N \rightarrow_l \lambda\langle x_1 \ldots x_{i-1}, x_{i+1} \ldots x_k\rangle.[x_i \mapsto N^n]M$
  if $1 \leq i \leq k$
- $(M_1|\ldots|(\lambda\langle x_1 \ldots x_k\rangle.M)^{n+1}|\ldots|M_n)N \rightarrow_l$
  $M_1|\ldots|\lambda\langle x_1 \ldots x_{i-1}, x_{i+1} \ldots x_k\rangle.[x_i \mapsto N^n]M|\ldots|M_n$,
  if $1 \leq i \leq k$
- $M^0 \rightarrow_l \perp$
- $\perp M \rightarrow_l \perp$
- $\perp^n \rightarrow_l \perp$
- $(M_1|\ldots M_m)^n \rightarrow_l M_1^n|\ldots|M_m^n$
- $M \rightarrow_l M' \Rightarrow M|N \rightarrow_l M'|N$
- $N \rightarrow_l N' \Rightarrow M|N \rightarrow_l M|N'$

- $M \rightarrow_l M' \Rightarrow MN \rightarrow_l M'N$

- $N \rightarrow_l N' \Rightarrow MN \rightarrow_l MN'$

$\xrightarrow{*}_l$ is the reflexive and transitive closure of $\rightarrow_l$.

Theorem 2 *(Strong normalization)*
Every reduction starting from a completely labelled term (M, I) terminates.

Proof: The proof is complicated but unsurprising. A detailed sketch of the proof is in an appendix. ∎

The following definitions are needed to access the label of a particular term. Define a map $min : Seq^\star \bigcup \{\infty\} \rightarrow \omega \bigcup \{\infty\}$ as

- $min(\vec{l}) =$ minimum of $\vec{l}$, if $\vec{l} \in Seq^\star$

- $min(\vec{l}) = \infty$, if $\vec{l} \notin Seq^\star$

Next, we define a reduction strategy $\rightarrow_\omega$ on the labelled terms as follows. This is done in a manner similar to the treatment of the unlabelled calculus. The following definitions, though simple are set out so that the proofs later will become intuitive. Define, by mutual recursion:
$Terms_1 ::= x \;||\; \lambda\langle x_1 \ldots x_k\rangle.M \;||\; MN || \; P^n$
$Terms_2 ::= M \;||\; M|N \;||\; Q^n$
where $M,\ N \in Terms_1 \bigcup Terms_2,\ P \in Terms_1,\ Q \in Terms_2$. The following definition is intended to capture the "number" of t_i's.

Define $len : Terms_2 \rightarrow Int$ as follows:

- $len(p) = 1$, if $p \in Terms_1$

- $len(p|q) = len(p) + len(q)$

It can be checked that this function is well-defined on the terms quotiented by the syntactic equality $\equiv$. The following definition is intended to capture the "position" of t_i in $t_1| \ldots t_n$. Define a partial function
$index : \omega \times Terms_2 \rightarrow Terms_1$ as follows:

- $index(n, p) =$ undefined if $len(p) \leq n \wedge \; len(p) \neq n$

- $index(1, p) = p$, if $p \in Terms_1$
- $index(n, p|q) = index(n, p)$, if $n \leq len(p)$
- $index(n, p|q) = index(n - len(p), q)$, if $len(p) \leq n \wedge len(p) \neq n$

Now, we have the machinery required to define the reduction strategy $\rightarrow_\omega$.

- $(\lambda\langle x_1 \ldots x_k\rangle.M)^{\vec{l}}N \rightarrow_{\omega\langle 1,\, i\rangle} \lambda\langle x_1 \ldots x_{i-1}, x_{i+1} \ldots x_k\rangle.[x_i \mapsto N^n]M$
  if $1 \leq i \leq k$, $\min(\vec{l}) = n + 1$
- $(\lambda\langle x_1 \ldots x_k\rangle.M)^{\vec{l}}N \rightarrow_{\omega\langle 1,\, i\rangle} \lambda\langle x_1 \ldots x_{i-1}, x_{i+1} \ldots x_k\rangle.[x_i \mapsto N]M$
  if $1 \leq i \leq k$, $\min(\vec{l}) = \infty$
- $index(s, (M_1|\ldots|(\lambda\langle x_1 \ldots x_k\rangle.M)^{\vec{l}}|\ldots|M_n)) = \lambda\langle x_1 \ldots x_k\rangle.M \Rightarrow$
  $(M_1|\ldots|(\lambda\langle x_1 \ldots x_k\rangle.M)^{\vec{l}}|\ldots|M_n)N \rightarrow_{\omega\langle s,\, i\rangle}$
  $M_1|\ldots|\lambda\langle x_1 \ldots x_{i-1}, x_{i+1} \ldots x_k\rangle.[x_i \mapsto N^n]M|\ldots|M_n$
  if $1 \leq i \leq k$, $\min(\vec{l}) = n + 1$
- $index(s, (M_1|\ldots|(\lambda\langle x_1 \ldots x_k\rangle.M)^{\vec{l}}|\ldots|M_n)) = \lambda\langle x_1 \ldots x_k\rangle.M \Rightarrow$
  $(M_1|\ldots|(\lambda\langle x_1 \ldots x_k\rangle.M)^{\vec{l}}|\ldots|M_n)N \rightarrow_{\omega\langle s,\, i\rangle}$
  $M_1|\ldots|\lambda\langle x_1 \ldots x_{i-1}, x_{i+1} \ldots x_k\rangle.[x_i \mapsto N]M|\ldots|M_n$
  if $1 \leq i \leq k$, $\min(\vec{l}) = \infty$
- $M \rightarrow_{\omega\sigma} M' \Rightarrow M|N \rightarrow_{\omega\sigma} M'|N$
- $N \rightarrow_{\omega\sigma} N' \Rightarrow M|N \rightarrow_{\omega\sigma'} M|N'$
  where $\sigma' = \langle first(\sigma) + len(N), second(\sigma)\rangle$
- $M \rightarrow_{\omega\sigma} M' \Rightarrow MN \rightarrow_{\omega\langle 1,\, \sigma\rangle} M'N$

Define:

- $\xrightarrow{*}_\omega$is the reflexive, transitive closure of $\rightarrow_\omega$
- $nf(\rightarrow_\omega) = \{M|\ M \notin dom(\rightarrow_\omega)\}$

We have the following corollary to the strong normalization theorem for completely labelled terms.

Lemma 59 *The reduction $\to_\omega$ restricted to completely labelled terms is strongly normalizing.*

The closed, completely labelled terms that are in $nf(\to_\omega)$ can be described completely. Define sets S_1, S_2, S_3, S_4 inductively as follows:

- S_1
  - $(\lambda\langle x_1 \dots x_k\rangle.M)^{\vec{l}} \in S_1$ if $FV(M) \subseteq \{x_1 \dots x_k\}$, $min(\vec{l}) = 0$
  - $M \in S_1 \Rightarrow MN \in S_1$, if N is closed.
  - $M, N \in S_1 \Rightarrow M|N \in S_1$
- S_2
  - $(\lambda\langle x_1 \dots x_k\rangle.M)^{\vec{l}} \in S_2$ if $FV(M) \subseteq \{x_1 \dots x_k\}$, $min(\vec{l}) = n+1$
  - $M, N \in S_1 \Rightarrow M|N \in S_2$
- S_3
  - $M \in S_1,\ N \in S_2 \Rightarrow M|N \in S_3,\ N|M \in S_3$
  - $M \in S_3,\ N \in S_2 \Rightarrow M|N \in S_3,\ N|M \in S_3$
  - $M \in S_3,\ N \in S_1 \Rightarrow M|N \in S_3,\ N|M \in S_3$
- S_4
  - $M \in S_3 \bigcup S_1 \Rightarrow MN \in S_4$
  - $M \in S_4, \Rightarrow MN \in S_4$

Note that the sets S_1, S_2, S_3 are pairwise disjoint.

Lemma 60 *(Classification of normal forms of closed terms)*
Let (M, I_M) be a completely labelled term. Then
$M \in nf(\to_\omega)$, M *closed* $\Rightarrow M \in S_1 \bigcup S_2 \bigcup S_3$

Proof: Structural induction. ∎

The reduction relations $\to_\omega$ and $\to$ are closely related. Since we need to talk about specific redices, we define the notion of a redex occurring at a position. This is done by structural induction.

- $\lambda\langle x_1 \dots x_k\rangle.M$ occurs at position $\langle 1, i\rangle$ in $\lambda\langle x_1 \dots x_k\rangle.M$, if $1 \leq i \leq k$.
- P occurs at position σ in $M \Rightarrow P$ occurs at position $\langle 1, \sigma\rangle$ in MN
- P occurs at position $\langle a, \sigma\rangle$ in $M \Rightarrow$
  - P occurs at position $\langle a, \sigma\rangle$ in $M|N$
  - P occurs at position $\langle a + len(N), \sigma\rangle$ in $N|M$

Many labels are omitted in the following discussion, for the sake of clarity.

Lemma 61 *(Relating $\rightarrow_\omega$ and $\rightarrow$)*

1. *Let $(M, I_M)\,(N, I_N) \in BC^\omega$. Then, $(M, I_M) \rightarrow_\omega (N, I_N) \Rightarrow [M \rightarrow N \ \vee \ M = N]$.*
2. *Let $M \rightarrow_\sigma N$. Let I_M be a labelling of M. Then, we have one of the following:*
   - *$(\exists I_N)\ [(M, I_M) \rightarrow_{\omega\sigma} (N, I_N)]$, or*
   - *The redex P occurring at σ has minimum label 0. Also, in this case, $(M, I_M) \in S_4$.*

Proof: Structural Induction. ∎

The relationship between the sets $S_i, i = 1 \dots 4$ and the convergence predicates $\Downarrow^{may}$ and $\Downarrow^{must}$ is described in the following lemma.

Lemma 62 *Let M be a term. Let (M, I_M) be a complete labelling of M.*

1. $\neg(M\Downarrow^{may}) \Rightarrow$
   $(\forall(N, I_N))\ [((M, I_M) \xrightarrow{*}_\omega (N, I_N) \wedge (N, I_N) \in nf(\rightarrow_\omega)) \Rightarrow (N, I_N) \in S_1]$
2. $\neg(M\Downarrow^{must}) \Rightarrow$
   $(\exists(N, I_N))\ [((M, I_M) \xrightarrow{*}_\omega (N, I_N) \wedge (N, I_N) \in S_4]$

Proof:

1. $(N, I_N) \in S_2 \bigcup S_3 \Rightarrow N\Downarrow^{may}$. The result follows from lemma 61.1.
2. $\neg(M\Downarrow^{must}) \Rightarrow$ there is an infinite reduction sequence
   $M = M_0 \rightarrow M_1 \rightarrow M_2 \dots M_k \dots$. Let (M, I_M) be a complete labelling. Since the reduction relation $\rightarrow_\omega$ is strongly normalizing, and using lemma 61.2 there exists a k such that, $(M, I_M) \xrightarrow{*}_\omega (M_k, I_{M_k})$, and $(M_k, I_{M_k}) \in S_4$. ∎

Approximable models

First, we abstract out the properties that are required for the proof of adequacy. The following definition is an extension of the definition of models with approximable application that is used for an analogous result for the lazy lambda calculus [50].

Definition 44 *$\langle D, Fun, Gr, \uplus, \star, \times, \diamond\rangle$ is approximable if*

1. *$\langle D, \uplus\rangle$ is a continuous algebra.*
2. *$\langle P(D), \star\rangle$ is a continuous algebra, $\{\!|\ |\!\} : (D \to D)_\perp \to P(D)$ is the singleton embedding.*
3. *$(Gr : P(D) \to D, Fun : D \to P(D))$ is a linear embedding-projection pair.*
4. *D has a least element $\perp$.*
5. *$\diamond$ is continuous.*
6. *There are continuous maps $[]_n : D \to D$, for each $n \in \omega$ satisfying:*
   (a) $d = \bigsqcup_{n\in\omega} d_n$
   (b) $d_0 = \perp$
   (c) $\perp \diamond d = \perp$
   (d) $d_{n+1} \diamond e \sqsubseteq [d \diamond e_n]_n$
   (e) $[[d]_n]_m \sqsubseteq [d]_{min(n,m)}$
7. *$\uplus : D \times D \to D$, is a continuous function that satisfies:*
   (a) $d \uplus e = e \uplus d$
   (b) $(d \uplus e) \uplus f = d \uplus (e \uplus f)$
   (c) $d \uplus d = d$
   (d) $[d \uplus e]_n \sqsubseteq [d]_n \uplus [e]_n$
   (e) $(d \uplus e) \diamond f \sqsubseteq (d \diamond f) \uplus (e \diamond f)$
8. *$\times : D \times D \to D$, is a continuous function that satisfies:*
   (a) $\perp \times \perp = \perp$
   (b) $(d \times \perp) \uplus \perp = d \times \perp$

(c) $d \times e = e \times d$

(d) $(d \times e) \times f = d \times (e \times f)$

(e) $[d \times e]_n \sqsubseteq [d]_n \times [e]_n$

(f) $d \times (e \uplus f) = (d \times e) \uplus (d \times f)$

(g)
- $[(d \uplus \perp \neq d) \wedge (e \uplus \perp \neq e)] \Rightarrow$
  $(d \times e) \diamond f \sqsubseteq ((d \diamond f) \times e) \uplus (d \times (e \diamond f))$
- $(d \uplus \perp \neq d) \Rightarrow$
  $(d \times e) \diamond f \sqsubseteq (d \times e) \diamond f \uplus d \diamond f$

9. $d \sqsubseteq e \uplus \perp \Rightarrow d \uplus \perp = d$

The initial solution to the recursive domain equation $D \simeq \overline{P}(D)$ satisfies the above conditions.

The semantics of the labelled calculus is defined as follows:

- $[\![(\perp, I)]\!]\ \rho\ = \perp$
- $[\![(x, I)]\!]\ \rho\ = [\rho(x)]_{min(I(x))}$
- $[\![(\lambda\langle x\rangle M, I)]\!]\ \rho\ = [Gr(d \mapsto [\![(M, I_M)]\!]\ \rho[x \mapsto d]\ )]_{min(I(\lambda\langle x\rangle M))}$
- $[\![(\lambda\langle x_1, x_2\rangle M, I)]\!]\ \rho\ =$
  $[Gr(\star[(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I_M)]\!]\ \rho[x_1 \mapsto d_1, x_2 \mapsto d_2]\ )),$
  $(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I_M)]\!]\ \rho[x_2 \mapsto d_1, x_1 \mapsto d_2]\ ))]])]_{min(I(\lambda\langle x_1, x_2\rangle M))}$
- $[\![(MN, I)]\!]\ \rho\ = [[\![(M, I_M)]\!]\ \rho\ ) \diamond\ [\![(N, I_N)]\!]\ \rho\ ]_{min(I(MN))}$
- $[\![(M|N, I)]\!]\ \rho\ = [[\![(M, I_M)]\!]\ \rho\ \times\ [\![(N, I_N)]\!]\ \rho\ ]_{min(I(M|N))}$

The following lemma establishes the relationship between $\perp$ and the syntactic classes S_1 and S_4.

Lemma 63 *Let (M, I) be a completely labelled term. Then,*

1. $(M, I) \in S_1 \Rightarrow [\![(M, I)]\!]\ \rho\ = \perp$
2. $(M, I) \in S_4 \Rightarrow [\![(M, I)]\!]\ \rho\ \uplus \perp = [\![(M, I)]\!]\ \rho$

Proof: Both proofs proceed by structural induction.

1. Recall that S_1 was defined inductively as:

   - $(\lambda\langle x_1 \dots x_k\rangle.M)^{\vec{l}} \in S_1$ if $FV(M) \subseteq \{x_1 \dots x_k\}$, $min(\vec{l}) = 0$
   - $M \in S_1 \Rightarrow MN \in S_1$, if N is closed.
   - $M, N \in S_1 \Rightarrow M|N \in S_1$

   Laws $6(a)$, $6(c)$, $8(a)$ of the definition of approximable models are used respectively for the cases.

2. Recall that S_3 was defined inductively as:

   - $M \in S_1,\ N \in S_2 \Rightarrow M|N \in S_3,\ N|M \in S_3$
   - $M \in S_3,\ N \in S_2 \Rightarrow M|N \in S_3,\ N|M \in S_3$
   - $M \in S_3,\ N \in S_1 \Rightarrow M|N \in S_3,\ N|M \in S_3$

   Laws $8(f)$, $8(c)$, $8(b)$, $8(d)$ of the definition of approximable models are used to prove the result. Recall that S_4 was defined inductively as:

   - $M \in S_3 \bigcup S_1 \Rightarrow MN \in S_4$
   - $M \in S_4, \Rightarrow MN \in S_4$

   Law $6(c)$ of approximable models is used to prove the result. ∎

Lemma 64 *Let (M, I) be a completely labelled term. Let (M_j, I_j), $j = 1 \dots n$, be all the terms such that $(M, I) \rightarrow_\omega (M_j, I_j)$. Then,*
$[\![(M,I)]\!]\ \rho \sqsubseteq [\![(M_1, I_1)]\!]\ \rho \uplus \dots \uplus [\![(M_n, I_n)]\!]\ \rho$.

Proof: Proof is by structural induction. Let (M, I) be a completely labelled term.
Case 0: $M = NP_1P_2$. Result follows from structural induction on NP_1, and the monotonicty of $\diamond$.

Case 1: $M = N|P$.
Let N_i, $i \leq i \leq n$ be all the possible terms such that $(N, I_N) \rightarrow_\omega (N_i, I_{N_i})$. Similarly, let P_j, $i \leq j \leq m$ be all the possible terms such that
$(P, I_P) \rightarrow_\omega (P_j, I_{P_j})$. From structural induction hypothesis,

- $[\![(N, I_N)]\!]\ \rho \sqsubseteq [\![(N_1, I_{N_1})]\!]\ \rho \uplus \ldots [\![(N_n, I_{N_n})]\!]\ \rho$

- $[\![(P, I_P)]\!]\ \rho \sqsubseteq [\![(P_1, I_{P_1})]\!]\ \rho \uplus \ldots [\![(P_m, I_{P_m})]\!]\ \rho$

Result follows from the bilinearity of $\times$ as given by laws $8(f)$ and $8(c)$ of approximable models.

Case 2: $M = (\lambda\langle x_1, x_2\rangle.N)P$. Let $M' = \lambda\langle x_1, x_2\rangle.N$

Let $min(I((\lambda\langle x_1, x_2\rangle.N)) = n + 1$. The two possible one step reductions($\rightarrow_\omega$) yield $\lambda x_1.[x_2 \mapsto P^n]M$ and $\lambda x_2.[x_1 \mapsto P^n]M$. By definitions, we have

$[\![(\lambda\langle x_1, x_2\rangle N, I)]\!]\ \rho\ =$

$[Gr(\star[(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I)]\!]\ \rho[x_1 \mapsto d_1, x_2 \mapsto d_2]\ )),$

$(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I)]\!]\ \rho[x_2 \mapsto d_1, x_1 \mapsto d_2]\ ))])]_{min(I(M')}$

Using the linearity of Gr and law $7(d)$ of approximable models

$[\![(\lambda\langle x_1, x_2\rangle N, I)]\!]\ \rho \sqsubseteq$

$[Gr([(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I)]\!]\ \rho[x_1 \mapsto d_1, x_2 \mapsto d_2]\ ))]_{min(I(M'))} \uplus$

$[Gr(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I)]\!]\ \rho[x_2 \mapsto d_1, x_1 \mapsto d_2]\ ))])]_{min(I(M'))}$

From definitions

$[\![((\lambda\langle x_1, x_2\rangle.N)P, I)]\!]\ \rho\ = [[\![(M, I)]\!]\ \rho \diamond [\![(P, I)]\!]\ \rho\ ]_{min(I(MN))}$.

Using law $7(e)$ of approximable models

$[\![((\lambda\langle x_1, x_2\rangle.N)P, I)]\!]\ \rho \sqsubseteq$

$[Gr([(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I)]\!]\ \rho[x_1 \mapsto d_1, x_2 \mapsto d_2]\ ))])]_{min(I(M'))} \diamond [\![(P, I)]\!]\ \rho$

$\uplus\ [Gr([(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I)]\!]\ \rho[x_2 \mapsto d_1, x_1 \mapsto d_2]\ ))])]_{min(I(M'))} \diamond [\![(P, I)]\!]\ \rho$.

From law $6(d)$ of approximable models, we have

$[Gr([(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I)]\!]\ \rho[x_1 \mapsto d_1, x_2 \mapsto d_2]\ ))])]_{min(I(M')} \diamond [\![(P, I)]\!]\ \rho$

$\sqsubseteq [\![(\lambda x_2.[x_1 \mapsto P^n]M, I)]\!]\ \rho$

From law $6(d)$ of approximable models, we have,

$[Gr([(d_1 \mapsto Gr(d_2 \mapsto [\![(M, I)]\!]\ \rho[x_1 \mapsto d_2, x_2 \mapsto d_1]\ ))])]_{min(I(M'))} \diamond [\![(P, I)]\!]\ \rho$

$\sqsubseteq [\![(\lambda x_1.[x_2 \mapsto P^n]M, I)]\!]\ \rho$. The result follows.

Case 3: $M = NP \wedge\ N = N_1 | \ldots | N_k$.

For notational convenience, assume $k = 2$. We have the following cases.

- All the N_is are of form $\lambda\langle x_1, \ldots x_k\rangle.N_i'$. Then, all the one step reductions are β-reductions involving P. The result follows from law $8(g)$ and the case above.

- N_i is not of form $\lambda\langle x_1, \ldots x_s\rangle.N_i'$, for $i = 1, 2$. Then, all the one-step reductions are of form

  - $N_1 \rightarrow_\omega N_1' \wedge\ (N_1|N_2)P \rightarrow_\omega (N_1'|N_2)P$
  - $N_2 \rightarrow_\omega N_2' \wedge\ (N_1|N_2)P \rightarrow_\omega (N_1|N_2')P$

  Result follows from induction hypothesis, observations $8(c), 8(f), 7(e)$.

- $N_1 = \lambda\langle x_1, \ldots x_s\rangle.N_1'$ and N_2 is not of this form. The one-step reductions are of form,

  - $N_2 \rightarrow N_2' \wedge\ (N_1|N_2)P \rightarrow (N_1|N_2')P$
  - $N_1 P \rightarrow N_1' \wedge\ (N_1|N_2)P \rightarrow (N_1'|N_2)$

  Let $Q_j,\ j = 1 \ldots t$ be such that $N_2 \rightarrow Q_j$. From structural induction hypothesis, $[\![N_2]\!]\ \rho \sqsubseteq [\![Q_1]\!]\ \rho\ \uplus \ldots [\![Q_t]\!]\ \rho$. Using law $8(c)$, $8(f)$, and structural induction hypothesis,
  $[\![N_1|N_2]\!]\ \rho \sqsubseteq [\![N_1|Q_1]\!]\ \rho\ \uplus \ldots [\![N_1|Q_t]\!]\ \rho$. Result now follows using law $8(g)$. ∎

The next lemma says that one can recover the meaning of a term in the original calculus by taking the least upper bound of the meanings of all the fully labelled terms.

Lemma 65 $[\![M]\!]\ \rho\ = \bigsqcup_{CL(M)}[[\![(M, I)]\!]\ \rho \mid I \in CL(M)]$

Proof: Structural induction. Note that the set $[[\![(M, I)]\!]\ \rho \mid I \in CL(M)]$ is directed.

- $M = x$. Result follows from law $6(a)$ of approximable models.

- $M = NP$.

$$\begin{aligned}
[\![M]\!]\ \rho &= \bigsqcup_{n\in\omega}[[\![NP]\!]\ \rho\ ]_n \\
&= \bigsqcup_{n\in\omega}[[\![N]\!]\ \rho\ \diamond [\![P]\!]\ \rho\ ]_n \text{ Indn. Hyp} \\
&= \bigsqcup_{n\in\omega}[(\bigsqcup_{I_N\in CL(N)}[\![(N, I_N)]\!]\ \rho\ ) \diamond (\bigsqcup_{I_P\in CL(P)}[\![(P, I_P)]\!]\ \rho\ )]_n \\
&= \bigsqcup_{n\in\omega}[\bigsqcup_{I\in CL(M)}[\![(N, I_N)]\!]\ \rho\ \diamond [\![P, I_P]\!]\ \rho\ ]_n \\
&= \bigsqcup_{I\in CL(M)}\bigsqcup_{n\in\omega}[[\![(N, I_N)]\!]\ \rho\ \diamond [\![P, I_P]\!]\ \rho\ ]_n \\
&= \bigsqcup_{I\in CL(M)}[\![(N, I_N)]\!]\ \rho\ \diamond [\![P, I_P]\!]\ \rho
\end{aligned}$$

- $M = \lambda\langle x_1, x_2\rangle N$. This involves the continuity of Gr and $\uplus$ and is omitted.

- $M = N|P$

$$\begin{aligned}
[\![M]\!]\ \rho &= \bigsqcup_{n\in\omega}[[\![N|P]\!]\ \rho\ ]_n \\
&= \bigsqcup_{n\in\omega}[[\![N]\!]\ \rho\ \times\ [\![P]\!]\ \rho\ ]_n \text{ Indn. Hyp} \\
&= \bigsqcup_{n\in\omega}[(\bigsqcup_{I_N\in CL(N)}[\![(N,I_N)]\!]\ \rho\ )\times\ (\bigsqcup_{I_P\in CL(P)}[\![(P,I_P)]\!]\ \rho\ )]_n \\
&= \bigsqcup_{n\in\omega}[\bigsqcup_{I\in CL(M)}[\![(N,I_N)]\!]\ \rho\ \times\ [\![P,I_P]\!]\ \rho\ ]_n \\
&= \bigsqcup_{I\in CL(M)}\bigsqcup_{n\in\omega}[[\![(N,I_N)]\!]\ \rho\ \times\ [\![P,I_P]\!]\ \rho\ ]_n \\
&= \bigsqcup_{I\in CL(M)}[\![(N,I_N)]\!]\ \rho\ \times [\![P,I_P]\!]\ \rho \qquad \blacksquare
\end{aligned}$$

Lemma 66 *Let M be a closed term. Then,* $\neg(M\Downarrow^{may})\Rightarrow [\![M]\!]\ \rho\ = \bot$

Proof: From lemma 65, $[\![M]\!]\ \rho\ = \bigsqcup_{CL(M)}[[\![(M,I)]\!]\ \rho \mid I \in CL(M)]$. Let (M,I) be any complete labelling of M. Let (M_j,I_j), $j = 1\ldots n$, be all the terms such that $(M,I)\xrightarrow{*}_\omega (M_j,I_j) \wedge (M_j,I_j) \in nf(\rightarrow_\omega)$. Then, from lemma 64
$[\![(M,I)]\!]\ \rho \sqsubseteq [\![(M_1,I_1)]\!]\ \rho\ \uplus\ \ldots\ \uplus\ [\![(M_n,I_n)]\!]\ \rho\ .$
From lemma 62, all the $(M_i,I_i) \in S_1$. From lemma 63, $(\forall 1 \le i \le n)\ [[\![(M_i,I_i)]\!]\ \rho\ = \bot$. Hence, the result. $\blacksquare$

Lemma 67 *Let M be a closed term. Then,* $\neg(M\Downarrow^{must})\Rightarrow [\![M]\!]\ \rho\ \uplus\ \bot = [\![M]\!]\ \rho$

Proof: From lemma 65, $[\![M]\!]\ \rho\ = \bigsqcup_{CL(M)}[[\![(M,I)]\!]\ \rho \mid I \in CL(M)]$. Let (M,I) be any complete labelling of M. Let (M_j,I_j), $j = 1\ldots n$, be all the terms such that $(M,I)\rightarrow_\omega (M_j,I_j)$. Then, from lemma 64
$[\![(M,I)]\!]\ \rho \sqsubseteq [\![(M_1,I_1)]\!]\ \rho\ \uplus\ \ldots\ \uplus\ [\![(M_n,I_n)]\!]\ \rho\ .$
From lemma 62, $(\exists I)\ [(M_i,I_i)\xrightarrow{*}_\omega (N,J) \wedge (N,J) \in S_4]$. From lemma 63, $[[\![(N,J)]\!]\ \rho \uplus \bot = [\![(N,J)]\!]\ \rho$. Hence, the result follows. $\blacksquare$

For the converse of the above two lemmas, we need the following lemma. In the proof, we use the following two facts about the model D.

1. $[(d \uplus \bot \neq d) \wedge\ (e \uplus \bot \neq e)] \Rightarrow$
   $(d \times e) \diamond f = ((d \diamond f) \times e) \uplus (d \times (e \diamond f))$

2. $(d \uplus \bot \neq d)\Rightarrow$
   $(d \times e) \diamond f \uplus d \diamond f = (d \times e) \diamond f$

3. $(d \uplus e) \diamond f = d \diamond f \uplus \; e \diamond f$

4. $(d \uplus e) \times f = (d \times f) \uplus (e \times f)$

5. $f \times (d \uplus e) = (f \times d) \uplus (f \times e)$

6. $(d \times e) \times f = d \times (e \times f)$

Lemma 68 *Let M be a closed, unlabelled term. Let M_i, $i = 1 \ldots n$ be such that $M \rightarrow M_i$. Then, $[\![M]\!]\;\rho \;= [\![M_1]\!]\;\rho \;\uplus \ldots [\![M_n]\!]\;\rho$.*

Proof: (Sketch) Proof is by structural induction.

- $M = NP_1P_2$. Follows from structural induction hypothesis on NP_1, and observation 3 above.
- $M = P_1|P_2$. Follows from structural induction hypothesis on P_1, P_2 and observation 4, 5, 6 above.
- $M = NP$, and $N = \lambda\langle x_1, \ldots x_k\rangle.N'$. Proof is almost identical to the corresponding case in lemma 64.
- $M = NP$, and $N = N_1|\ldots N_k$. Proof is almost identical to the corresponding case in lemma 64.

The following lemma is the converse of lemma 66 and lemma 67.

Lemma 69 *let M be a closed term. Then,*

- $M{\Downarrow^{may}} \Rightarrow [\![M]\!]\;\rho \;\neq \bot$
- $M{\Downarrow^{must}} \Rightarrow [\![M]\!]\;\rho \;\uplus \bot \neq [\![M]\!]\;\rho$

The following theorem is an immediate consequence of lemma 69, lemma 66, lemma 67.

Theorem 3 *(Adequacy)*
Let M, N be closed terms. Then, $[\![M]\!]\;\rho \;\sqsubseteq [\![N]\!]\;\rho \;\Rightarrow M \preceq N$

3.6 Conclusions

The work in this chapter represents part of a growing interest in higher-order process calculi. This chapter studied a *higher-order* process calculus, a restriction of one due to Boudol [10], and developed an abstract, mathematical model for it. The model was constructed domain theoretically and reflects a certain conceptual viewpoint about observability. The main new technical tool that was used was a new powerdomain construction. We showed that the powerdomain can be given additional algebraic structure that allows one to model concurrent composition, in the same spirit that Plotkin's powerdomain can have a continuous binary operation defined on it to model choice. The model was shown to be adequate with respect to the operational semantics.

It would be interesting to know what it takes to make the calculus fully abstract. Given that the language has concurrency "naturally" built into it one might expect that one would get full abstraction by adding a simple convergence tester; unlike the case of the lazy lambda calculus where one needed a parallel convergence tester [4]. This, however, seems unlikely though there is no definitive answer, as yet. It would also be interesting to understand the structure of the powerdomain more clearly: in particular, a modal characterization of the powerdomain will help to relate it more precisely to the standard constructions, for which such characterizations are known [1,19]. A direct application would be that the adequacy proof can be simplified enormously and carried out in the style of an adequacy proof for the lazy lambda calculus [4].

Also, this chapter studied a restricted version of the full γ-calculus. The restrictions made simplified certain aspects of the calculus: for example, deadlock is not possible, but it preserves much of the complexity. In particular both concurrency and nondeterminism still exist. The investigations of the extensions of these ideas to the full calculus will be the subject of future work.

The ideas underlying the powerdomain construction described in this chapter seem to be applicable in various other contexts. The same idea of "extensional ordering" among functions underlies the powerdomain used in the next chapter on the semantics of concurrent logic programming. For a different application of the powerdomain construction , consider the work on the semantics of the simply typed lambda calculus with a choice operator at ALL types. The traditional semantics of linear function types [18,21], is correct for a "call by value" operational semantics. If one attempts to model a "call by name" evaluation strategy, one runs into the kind of problems that motivated the powerdomain construction described in this chapter. More precisely, consider the simply typed lambda calculus with an OR at all types, with "call by name" order of evaluation. Also, assume that all observations are of first order values only. Consider the terms, $\lambda x.[1\ or\ 2]$ and $\lambda x.1\ or\ \lambda x.2$. Intuitively, these terms are intersubstitutable in any first order context. However, in the standard powerdomain constructions, the denotations of these terms are different. Note the striking similarity to the example that motivated the powerdomain construction: the identification of the terms $\lambda x.[\Omega\ or\ \lambda y.\Omega]$ and $\lambda xy.\Omega\ or\ \lambda x.\Omega$ by the operational semantics of the chapter. The key point is that this operational semantics demands that lambda abstraction distribute over choice. The "extensional powerdomain" presented in this chapter is geared to provide a solution to this problem.

Chapter 4

Closure Operator Semantics for Concurrent Constraint Logic Programming

4.1 Introduction

Logic programming generalizes Kahn's model of static dataflow 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 proof-theoretic basis based on viewing computations as proofs in first order logic, carried out with a single inference rule, namely resolution. Pragmatic 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. For the purposes of this chapter, the key insight is the use of logic variables for synchronization through blocking reads: a process that reads an uninitialized logic variable waits until the variable is instantiated. In the presence of this kind of synchronization, the correspondence of programs to proofs in first order logic becomes tenuous. This is the motivation for investigations into an abstract semantics for logic programs. Motivated by Kahn's elegant model of static determinate 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 chapter[1] is intended to be a piece in this program. The main contribution of the chapter is a fully-abstract semantics of concurrent constraint programming languages similar to Flat GHC [63]. The language has synchronization capabilities and committed choice (also called "don't know non-determinism").

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 or computation paths. The semantics follows the techniques of chapter 2, and can be viewed as the extension of these ideas to an indeterminate setting. The semantics makes no atomicity assumptions and handles infinite behaviors. Furthermore, the semantics works correctly only under the assumption of AND-fairness. The structure used to model indeterminacy is a variant [26] of the standard powerdomains used to model indeterminacy in imperative languages [52,64].

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 [35]. 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 operational and the denotational 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. There is a fully abstract semantics for a language based on the Ask-Tell paradigm [57]. 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 [17,57] do not handle infinite computations. There is also an extant semantics [13] that is fully-abstract and handles infinite computations. This semantics is based on keeping track of the communication actions of processes along every computation path.

The significant difference between this work and the papers cited above arises from the different motivations. Previous work [17,57,13] can be viewed as starting out with

[1]The material in this chapter is drawn from joint work with Vasant Shanbhogue [29]

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 modeled by the semantics is a significant and non-trivial subset of the observations modeled in previous work.

The rest of this chapter is organized 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 dataflow, as an aid to understanding the semantics. The language is powerful enough to code typical programs, such as the short-circuit protocol. Next, we give a detailed description of our notion of tests. In the next section, we describe the domain theoretic structures needed for the denotational model. Then, we describe the denotational semantics. In the next section, we present the proof of full-abstraction. In the final section, we sketch the extra structures needed to model the full language.

4.2 Operational semantics

4.2.1 Syntax and reduction rules

In this section, we define the language studied for the major portion of the chapter. The language is very similar to the language flat GHC [63]. The basic difference with flat GHC, 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 function 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 chapter.

The presentation of the syntax and the reduction relation follows standard notation [63]. A concurrent constraint logic program is a set of guarded clauses. 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. We assume that the heads A is made up of *distinct* variables. The exact structure of the guard predicates is described in a later subsection. For presenting the operational semantics, we need the following syntactic categories.

$x, L \in$ Id = countable set of identifiers
$f \in$ Functions = set of function symbols
$p \in$ Pred = set of predicate symbols
$t \in$Terms = Variables or a function symbol of arity n applied to n terms
$A, B \in$Atoms = $p(t_1, \ldots t_n)$, if p n ary predicate, t_i are terms
$E \in$ Alias-set ::= $\{t_1, ..., t_n\}$ $\rho \in$ Environment ::= $\phi|\{E_1, ..., E_n\}$

The transition system is defined in terms of a input match and try function. The input match function describes the matching of the actual parameters of a predicate call in a program to the formal parameters and uses unification. The try function describes the selection of a particular clause definition for a predicate name from the set of clauses defining the predicate name.

Unification

As in Chapter 2, we use the unification algorithm, without the occurs check. This has been extensively studied [61,12,25,70]. 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 45 $\rightsquigarrow$ *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 $\{f(t_1, ..., t_n), f(t'_1, ..., t'_n)\} \subseteq A \in \rho$ then $\rho \rightsquigarrow \rho \cup \{\{t_1, t'_1\}, ..., \{t_n, t'_n\}\}$.*
3. *If $\{f(t_1, ..., t_n), g(t'_1, ..., t'_n)\} \subseteq A \in \rho$ 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 $\rightsquigarrow$. It can be shown [61], that $\overset{*}{\rightsquigarrow}$ is Church-Rosser and Noetherian(terminating). In the rest of the chapter, we will usually not be concerned with the explicit details of the algorithm; in particular, we will not distinguish between syntactic environments and their reduced forms.

Input matching

Input matching is formalized as a function *match* that takes two terms and an environment as arguments. The type of the *match* function is $Terms \times Terms \times Environment \rightarrow \{Environments, fail\}$. The definition is by structural induction on terms.

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

$$match(\vec{x}, \vec{u}) = \begin{cases} fail, \text{ if } \rho \bigcup \{(x_1, u_1), \ldots (x_n, u_n)\} = error \\ \rho \bigcup \{(x_1, u_1), \ldots (x_n, u_n)\}, \text{ otherwise} \end{cases}$$

Match can be easily extended to a function from $Atoms \times Atoms \times Environment$ to $\{Environments, fail, susp\}$

Try function

The try function chooses a particular definition for the predicate name from the set of definitions for the predicate. The intuition is as follows. Consider a procedure call of form $p(t_1, \ldots p_n)$ and a syntactic environment ρ. Then, the procedure p can be replaced by the body of the definition $p(x_1 \ldots x_n) \leftarrow G|B_1, \ldots B_n$, if the syntactic environment ρ is such that the evaluation of the guard G "succeeds".

The description of the try function requires the formalization of the evaluation of a guard predicate in a syntactic environment. For this, 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, and the addition of i to the head of the sequences by $[i|s]$. Let $t = f_i^n(t_1, \ldots t_n)$ be any term. Then, $t \uparrow s$ is defined inductively by the two rules :$t \uparrow 0 = f_i^n$ and $t_i \uparrow s = g \Rightarrow t \uparrow [i|s] = g$. An expression of form $t \uparrow s$ is "evaluated" in an environment ρ by the following rules:

(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. It can be proved that this seemingly arbitrary choice does not affect the results of the evaluation of $\langle e, \rho \rangle$, in the interesting cases.

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

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

Given the notion of the evaluation of a term in an environment, we can describe the evaluation of a guard predicate in a syntactic environment. 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 $=:=, \leq, \neq:\neq$,

We define below the evaluation of a primitive guard predicate in a syntactic environment ρ. The meanings of the other primitive guards are defined similarly.

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

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

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

The try function can now be defined. This definition uses the evaluation of guard predicates described above.

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

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

Rewrite rules

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 the configuration with inconsistent environment $fail$. $\{C_i\}$ is a multiset of atoms.

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

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

The suspend 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** in the literature. We follow the definitions of previous work [63]. 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.

4.2.2 Observables and Operational equivalence

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 4.5.

This section is organized as follows. We first define the notion of finite observations of environments, and use it to define finite observations of processes. 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. This theory is closely related to previous work in the context of process calculi [20]. A notion of tests is then defined. The tests that we allow correspond roughly to placing the process in arbitrary contexts. The notion of test allows us to define the idea of operational equality of processes: two processes are operationally equivalent if and only if they pass the same tests.

Finite observations of environments

Finite observations of environments correspond to looking at the tree structure of finitely many variables to a finite depth. Thus, in the information ordering of environments, the observations that we are considering correspond to finite (in the sense of domain theory) approximations to environments. This is the motivation behind the following definitions. These definitions give a way of syntactically capturing finite environments. This is done by first describing a syntactic notion "finite" terms and then extending the idea to syntactic environments.

We formalize the idea of looking at the finite approximations to the tree structure of variables first. This is done by defining a class of finite terms that correspond to the finite elements of the value domain V. In the definition below we use the symbol Ω to model "no information". The set of finite terms *Finterms* is defined inductively as the smallest set containing Ω and satisfying the following closure condition: $t_1 \ldots t_n \in Finterms \Rightarrow f(t_1 \ldots f_n) \in Finterms$, where f is an n-ary function symbol.

The above definition can be extended to syntactic environments. Recall that the syntactic environment was presented as a set of alias sets. 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. The rule is $\rho \cup \{A_1 \cup \{\Omega\}\} \leadsto \rho \cup \{A_1\}$. 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 can define the notion of finite observations of environments. 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 propositions true in a given syntactic environment. $\models$ is written infix.

Definition 46 $\rho \models t \sqsubseteq \rho(x)$, *if* ρ *inconsistent or* $t \sqsubseteq \rho(x)$.

Note the handling of inconsistent environments. The inconsistent environment satisfies any constraint. This is motivated by the constraint view of the environment [60]: an environment that has inconsistent constraints imposed on it logically implies any constraint.

Finite observations of processes

In this subsection, finite observations of processes are defined. There are two aspects of the behavior of processes that are of interest: the resulting environment and the question of termination. The previous subsection defined finite observations of environments. For handling termination, define $Term = \{terminated, donotcare\}$: *terminated* is intended to indicate that the execution terminates, and *donotcare* is intended to indicate that execution may or may not terminate. For example, as a first approximation, we can say that a process executed in an initial environment satisfies a pair $\langle terminated, p\rangle \in Term \times Primobs$, if the execution of the process terminates in a resulting environment that satisfies p. Similarly, we can say that a process executed in an initial environment satisfies a pair $\langle donotcare, p\rangle \in Term \times Primobs$, if there is a configuration at some finite stage of execution such that the environment in this configuration satisfies p. Since the information in the environments increases monotonically with program execution, we can alternatively state this as: for all but finitely many configurations, the environments satisfy p. Such an approach would work nicely if the processes are determinate. For indeterminate processes, it is more natural to think of observations as disjunctions of pairs from $Term \times Primobs$: the intuition is that every execution sequence of the program satisfies one of the disjuncts at a finite stage. This is the approach that is adopted in the rest of this section.

Define OBS as the the set of all expressions generated from $Term \times Primobs$ using the boolean connectives $\wedge$ and $\vee$. We will denote members of OBS by o, o_1 and so on. $\vdash \subseteq CONF \times OBS$, 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. Informally, $\langle C, \rho\rangle \vdash o$ for an observation $o = \langle p, term\rangle$ means the following: every computation path c_i of $\langle C, \rho\rangle$ contains a configuration $\langle C_i, \rho_i\rangle$ such that $\rho_i \models p$. Furthermore, if $term$ is $terminated$, $C_i = true$ or $C_i = false$; that is, the computation has terminated. More precisely, we have the following definition:

Definition 47 *(Definition of $\vdash$)*

1. $C = \langle true, \rho\rangle \wedge \rho \models p \Rightarrow \langle C, \rho\rangle \vdash \langle terminated, p\rangle$
2. $\rho \models p \Rightarrow \langle C, \rho\rangle \vdash \langle donotcare, p\rangle$
3. $C = fail \Rightarrow \langle C, \rho\rangle \vdash \langle terminated, p\rangle$
4. $\langle C, \rho\rangle \vdash o_1$ *or* $\langle C, \rho\rangle \vdash o_2 \Rightarrow \langle C, \rho\rangle \vdash o_1 \vee o_2$
5. $\langle C, \rho\rangle \vdash o_1$ *and* $\langle C, \rho\rangle \vdash o_2 \Rightarrow \langle C, \rho\rangle \vdash o_1 \wedge o_2$
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* $o = \langle p, t\rangle \in OBS \times Term$ *be such that all variables occurring in* p *occur in* $\langle C, \rho\rangle$. *Furthermore, let* $\langle C_i, \rho_i\rangle \vdash o$. *Then,* $\langle C, \rho\rangle \vdash o$

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. Thus, for example, only a failed computation can pass a test of the form $\langle terminated, x = 1\rangle \wedge \langle terminated, x = 2\rangle$. 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.

Tests

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.

4.3 Denotational Semantics

This section describes the denotational semantics of the language. First, we describe the mathematical structures used to model the language. In the next subsection, we describe the formal semantics of the language.

Domain-theoretic facts

In this section, the domain-theoretic tools needed for the semantics are developed. In the first subsection, the construction of the valuedomain is described. In the second subsection, a variant of closure operators used in the semantics is defined. In the final subsection, the structures used to model indeterminacy are presented.

Domain of values

The presentation of this subsection follows Chapter 2. 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 [37]. 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 written $|X|$ and the size of a sequence s is written $|s|$.

Definition 48 *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 $\top$. The domain W is defined as follows. Let $Atom$ be a given set of function symbols. Let $A = Atom \cup \{\Omega\} \cup \{f_i^n\}$ where Ω stands for the undefined element.

Definition 49 *An element of W is a function $F : t \rightarrow A$ where t is a non-empty tree. The function f satisfies*
$\forall s \in t.br(s) = n \Rightarrow F(s) = f$, 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 function 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 $\top$. $\top$ is the model for inconsistent constraints. It is straightforward to check that V is a complete algebraic lattice with finite $\top$. As in Chapter 2, the $\top$-strict product domain of algebraic lattices with finite $\top$, D_1 and D_2, is denoted $D_1 \times^{\top} D_2$. The space of semantic environments is denoted by $ENV = \Pi_x^{\top} V_x$, where x ranges over variable names.

Function space

This section studies a variant of closure operators, equipped with extra structure to detect termination. Denote by $\mathbf{2}$, the domain with two elements $\{\perp, \top\}$, ordered as $\perp \sqsubseteq \top$.

Definition 50 *Let D be an algebraic lattice with finite $\top$. $[D \xrightarrow{c} D \times \mathbf{2}]$ is the set defined as follows. The elements f of $[D \xrightarrow{c} D \times \mathbf{2}]$ are continuous functions from D to $D \times \mathbf{2}$, satisfying $f(x) = \langle y, t\rangle \Rightarrow [x \sqsubseteq y \wedge f(y) = \langle y, t\rangle]$*

Writing Π_1 for the projection onto the first argument, the above definition implies that $\Pi_1 \circ f$ is a closure operator on D if $f \in [D \xrightarrow{c} D \times \mathbf{2}]$. The elements of $\mathbf{2}$ in the second component of the result are used as "termination signals". If the second argument is $\top$, the computation is terminated and if the second argument is $\perp$, the computation is deadlocked or non-terminating. This correspondence is made precise in the proof that relates the abstract and operational semantics.

A couple of examples will help to make the ideas 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 $\Pi_i^{\top} V_i$, 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:

$$\begin{aligned}
\mathcal{E}[\![f]\!]\vec{a} &= \langle a_1 \ldots a_n, f(a_1 \ldots a_n)\rangle, \text{ if } a_{(n+1)} = \perp \\
\mathcal{E}[\![f]\!]\vec{a} &= \top, \text{ if } a_{(n+1)} \notin W_f \\
\mathcal{E}[\![f]\!]\vec{a} &= \langle a_1 \sqcup b_1 \ldots a_n \sqcup b_n, f(a_1 \ldots a_n) \sqcup a_{(n+1)}\rangle, \text{ if } a_{(n+1)} = f(b_1, \ldots b_n)
\end{aligned}$$

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

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

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

Elements of $[D \xrightarrow{c} D \times \mathbf{2}]$ share some of the nice properties of the space of closure operators defined in Chapter 2. In particular, $f \in [D \xrightarrow{c} D \times \mathbf{2}]$ can be equivalently characterized 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 $x \sqsubseteq y$.

Definition 51 *Let D be an algebraic lattice with finite $\top$. A subset S of $D \times \mathbf{2}$ is said to be a valid set of fixpoints if it satisfies:*

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 an element of $[D \xrightarrow{c} D \times \mathbf{2}]$ is a valid set of fixpoints. Furthermore, given a valid set of fixpoints, we can recover the corresponding function. Let S be a valid set of fixpoints. Then, f_S is defined as follows. Let $x \in D$. Define $f_S(x) = \sqcap S_x$.

We can define a parallel composition operation $||$. Let f, g be elements of $[D \xrightarrow{c} D \times \mathbf{2}]$. Let $x \in D$.

Definition 52 *Define $h = f||g$ as follows: let $y = \bigsqcup_i \{((\Pi_1 \circ g) \circ (\Pi_1 \circ f)))^i(x) | i\}$. Thus, $f(y) = \langle y, t_1\rangle, g(y) = \langle y, t_2\rangle$. Define $h(x) = \langle y, t_1 \sqcap t_2\rangle$.*

It is easy to see that $h = f||g$ defined as above is a member of $[D \xrightarrow{c} D \times 2]$. Note that $h(x)$ is the least solution(denoted by lcs) of a system of equations as follows:

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

This yields an elegant characterization 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 70 $\langle x, t \rangle \in Fix(f||g) \Leftrightarrow (\exists \langle x, t_1 \rangle \in Fix(f))\ (\exists \langle x, t_2 \rangle \in Fix(g))\ [t = t_1 \sqcap t_2]$

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

Lemma 71 $||$ *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 $\top$. Let f_i be an element of $[D_0 \times^{\top} D_i \xrightarrow{c} D_0 \times^{\top} D_i \times 2]$. We can define the "shared" parallel composition of $f_1 \ldots f_n$. 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 $ENV \times^{\top} V$. To compute the parallel composition 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 $||_{D_0}\langle f_1 \ldots f_n \rangle$. Let Id_i denote the identity function on D_i. Define

$$f_i' = (\Pi_0 \circ f_i \circ \langle \Pi_0, \Pi_i \rangle) \times^{\top} Id_1 \ldots \Pi_i \circ f_i \circ \langle \Pi_0, \Pi_i \rangle \times^{\top} Id_n$$

Set $||_{D_0}\langle f_1 \ldots f_n \rangle \in [D_0 \times^{\top} D_1 \times^{\top} \ldots D_n \xrightarrow{c} D_0 \times^{\top} D_1 \times^{\top} \ldots D_n \times 2] = ||\{f_1' \ldots f_n'\}$.

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 72 *Given $f, g \in [D \xrightarrow{c} D \times 2]$, the ordering is described as:*

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

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

For the denotational semantics we need a notion of guarded closure operators. Let $Bool == \{\bot, tt, ff, \top\}$ 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

$$(s|f)v = \begin{cases} f(v) \;\; if \; s(v) = tt \\ \langle \top, \top \rangle \;\; if \; ff \sqsubseteq s(v) \\ \langle v, \bot \rangle \;\; otherwise \end{cases}$$

It is easy to check that the monotonicity and the continuity of the operation defined above.

The following lemma sets up the usual cpo framework for the semantics of recursion.

Lemma 73 *Let D be an algebraic lattice with finite $\top$. 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 upper bounds of chains are got by the usual pointwise limit. The prime elements of the function space are functions of the form $f_{a,b}$, where

$a \sqsubseteq b$ are finite elements and defined by

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

Finite elements are lubs of finitely many prime elements. ∎

Powerdomain construction

In this subsection, the machinery to handle indeterminacy is developed. For motivation, note that the key observation underlying the semantics for the determinate language in Chapter 2, was that computations could be thought of as closure operators. One way of thinking about indeterminate computations is to view them as a set of computations. Thus, the aim is to set up machinery for handling sets of functions. The development of these tools is is the primary aim of this section. This is done using the powerdomain construction used in Chapter 3.

Let $B(D)$ be the basis of D. The basis of the Smythe powerdomain of D, denoted by $P_S(D)$ is the finite powerset of the basis elements, $P_{fin}(B(D))$. These elements are ordered as $\{d_1 \dots d_n\} \sqsubseteq \{e_1 \dots e_m\} \Rightarrow (\forall 1 \leq j \leq m)\ (\exists 1 \leq i \leq n)\ [d_i \sqsubseteq e_j]$. The Smythe powerdomain [64] of D, denoted $\overline{P_S(D)}$, is the ideal completion of $P_S([D \xrightarrow{c} D \times 2])$: the elements of $\overline{P_S(D)}$ are downward closed and directed subsets S ordered by the subset ordering. $\overline{P_S(D)}$ can be made into a continuous algebra [64] with a union operation $\uplus$, defined as: $S_1 \uplus S_2 = \{s_1 \bigcup s_2 | s_1 \in S_1, s_2 \in S_2\}$. Note that the operation $\uplus$ is idempotent, commutative and associative.

The powerdomain construction on the function space is based on an "extensional" ordering among sets of functions. Define a set-theoretic function *App*, with domain $P_{fin}(B([D \xrightarrow{c} D \times 2])) \times D$ and range $\overline{P_S(D)}$:

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

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

Definition 53 *$P([D \xrightarrow{c} D \times 2])$ is the preorder with carrier $P_{fin}(B([D \xrightarrow{c} D \times 2]))$. The ordering relation is defined as follows.Let $F = \{f_1 \dots f_n\}, G = \{g_1 \dots g_m\}$. Then $F \sqsubseteq G$ if $F \sqsubseteq G \Rightarrow [(\forall x)\ App(F, x) \sqsubseteq App(G, x)]$.*

$\overline{P([D \xrightarrow{c} D \times 2])}$ is the ideal completion of $P([D \xrightarrow{c} D \times 2])$. $\overline{P([D \xrightarrow{c} D \times 2])}$ can be made into a continuous algebra [64] with a union operation $\uplus$, defined as: $S_1 \uplus S_2 = \{s_1 \bigcup s_2 | s_1 \in S_1, s_2 \in S_2\}$. $\uplus$ is idempotent, commutative and idempotent. Also, *App* as defined above can be extended to a continuous function from $\overline{P([D \xrightarrow{c} D \times 2])} \times D$ to $\overline{P_S(D)}$.

There is a singleton embedding function $\{\!|\ |\!\}$ with domain $[D \xrightarrow{c} D \times 2]$ and range $\overline{P([D \xrightarrow{c} D \times 2])}$, defined on the basis of $[D \xrightarrow{c} D \times 2]$ as: $\{\!| f |\!\} = \{f\}$. It can be easily checked that $\{\!|\ |\!\}$ is monotone. Thus, $\{\!|\ |\!\}$ can be extended uniquely to a continuous function.

In this section, we extend the semantic functions defined on $[D \xrightarrow{c} D \times 2]$ to $\overline{P(D)}$. The basic intuition is to extend the definitions "linearly": *i.e.* extend pointwise to sets of functions definitions on single functions. The basic strategy is to define the functions on the basis of $\overline{P(D)}$, namely $P(D)$. Proving that the definition is monotone on $P(D)$ ensures that there is a unique continuous extension to $\overline{P(D)}$.

The parallel composition operation on $\overline{P([D \xrightarrow{c} D \times 2])}$ can be defined as follows. The following definition is on the basis elements $P([D \xrightarrow{c} D \times 2])$.

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

Lemma 74 $||$ *as defined above is monotone in both arguments.*

Proof: Let $F = \{f_1 \ldots f_n\}$, $G = \{g_1 \ldots g_m\}$ and $G \sqsubseteq F$. Let $H = \{h_1 \ldots h_k\}$. 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

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

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

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

$g_j || h_k(x) \sqsubseteq g_j || h_k(y) \sqsubseteq f_i || h_k(y) = \sqsubseteq f_i || h_k(x)$
since $x \sqsubseteq y$. 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 $F \times^{\top} Id_{D_2}$, where $F \in \overline{P([D_2 \xrightarrow{c} D_2 \times \mathbb{2}])}$. We define a continuous function $\times^{\top} Id_{D_2}$. The domain of $\times^{\top} Id_{D_2}$ is $\overline{P([D_1 \xrightarrow{c} D_1 \times \mathbb{2}])}$ and its range is $\overline{P([D_1 \times^{\top} D_2 \xrightarrow{c} D_1 \times^{\top} D_2 \times \mathbb{2}])}$. This is written postfix for readability. As usual, we define $s \times^{\top} Id_{D_2}$, for $s \in P([D_1 \xrightarrow{c} D_1 \times \mathbb{2}])$. Let $s = \{f_1 \dots f_n\}$. Then,

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

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

We can also define a continuous operator $s|$ on $\overline{P([D \xrightarrow{c} D \times \mathbb{2}])}$, where s is a continuous function from ENV to $Bool$. As usual, we define s only on the finite elements of $\overline{P([D \xrightarrow{c} D \times \mathbb{2}])}$:

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

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 $\overline{P([D \xrightarrow{c} D \times \mathbb{2}])}$.

The Semantics

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

- Terms : $[ENV \times^{\top} V \xrightarrow{c} ENV \times^{\top} V \times \mathbb{2}]$.
- n-ary predicates p : element of $\overline{P([\Pi^{\top}_{1 \le i \le n} V_i \xrightarrow{c} \Pi^{\top}_{1 \le i \le n} V_i \times \mathbb{2}])}$
- Atoms and sequences of atoms: $\overline{P([ENV \xrightarrow{c} ENV \times \mathbb{2}])}$.

The definition of the denotations of terms is by structural induction on terms. The definition will encompass meanings of sequences of terms, $\mathcal{E}[\![\langle t_1 \dots t_n \rangle]\!]$. The denotation of a sequence of terms of length n is a closure operator on the space $ENV \times^{\top} \Pi^{\top}_n V$. 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 : $\mathcal{E}[\![x]\!]\ \langle env, a\rangle = \langle\langle env[x \mapsto b], b\rangle, \top\rangle$ $where\ b = env(x) \bigsqcup a$

- 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,

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

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 modeled 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 operation in defining the termination signal of the result of the parallel composition operation in definition 52 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.

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

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 $Bool$.

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

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 modeled 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 made $\mathcal{E}[\![G]\!]$ evaluate to true.

Failure: This happens operationally when the environment has sufficient constraints to make the guard predicate fail. Semantically, this is modeled 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 modeled 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 $[\Pi^{\top}_{1\leq i\leq n} V_i \xrightarrow{c} \Pi^{\top}_{1\leq i\leq n} V_i \times \mathbf{2}]$. 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 uninitialized 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 $\Pi^{\top}_n V$. The elements of $\mathcal{E}[\![p]\!]$ correspond to the different choices of execution. Furthermore, each element(choice) has the same intuitive 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. Given the 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.

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

The first item above was modeled 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

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

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 environment and the value returned by execution of p are equal.

4.4 Relating the two semantics

In this section, we prove that the operational and denotational views of programs coincide. The proof can be viewed as the extension of the full-abstraction proof of Chapter 2 to an indeterminate setting. As before, the proof proceeds in three stages.

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. The proof proceeds as follows. We associate denotations with operational configurations, in a style similar to that adopted in Chapter 2. Thus, it is intended that $\mathcal{M}[\![\langle C,\rho\rangle]\!]$ represents the effect of the complete computation on the 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 $\mathcal{M}[\![conf_i]\!]$. Then, we show that $S = \uplus_i S_i$. As a corollary, we show if a process passes a finite test, the set of possible results produced by the denotational semantics "attests" to 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. Analogous properties have been termed computational adequacy [40]. As in chapter 2, 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 and is motivated by the proof in Chapter 2. Indeed, the proof generalists the techniques of Chapter 2 to an indeterminate setting.

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 "attests" 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 reasoning about operational equality. This is called adequacy [16]. 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 [53] [40].

4.4.1 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 results 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.

The proof requires a translation of the syntactic environment into a set of equations. We formalism 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 $\overline{P([ENV \xrightarrow{c} ENV \times 2])}$.

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

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

$$Dom(env) \cap (Var - Dom(\rho)) = \emptyset \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (I)$$

$Dom(env)$ 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 75 *Let the defining clauses for p be given by*

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

$$\cdots$$

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

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* $p_i(\vec{x}) \leftarrow G_i | C_i$.

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

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

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*

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

$$\cdots$$

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

Consider the configuration $\langle p(\vec{t}), \rho \rangle$. *Furthermore, assume that* $\mathcal{M}[\![\langle p_i(\vec{t}), \rho \rangle]\!] env_{\bot} = \top$. *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 $\overline{P_S(ENV)}$, $S \uplus \{\top\} = S$. $\blacksquare$

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 76 *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:

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

where $\mathcal{E}[\![p]\!] = \lambda\vec{a}.\Pi_{\vec{x}} \circ \mathcal{E}[\![G|C]\!] \circ [env_{\bot}[\vec{x} \mapsto \vec{a}]$ 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 $Id_{ENV} \times^{\top} \mathcal{E}[\![G|C]\!]$ 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:

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

Since we are interested only in the effect on the variables in ρ, the function of interest is $\Pi_{ENV} \circ \mathcal{M}[\![\langle G|C, \rho \bigcup \{\vec{x} = \vec{t}\,\rangle]\!]$. $\Pi_{ENV} \circ (Id_{ENV} \times^{\top} \mathcal{E}[\![G|C]\!]) || \{\vec{x} = \vec{t}\} || EQ(\rho)$ can be rewritten as $\Pi_{ENV} \circ (Id_{ENV} \times^{\top} \mathcal{E}[\![p]\!]) || (\{\!| \mathcal{E}[\![\vec{t}]\!] |\!\}) \circ \langle Id_{ENV}, \vec{\bot}\rangle$ because the effect of the equation $\{\vec{x} = \vec{t}\}$ is captured by the parallel composition of $Id_{ENV} \times^{\top} \mathcal{E}[\![p]\!]$ and $\{\!| \mathcal{E}[\![\vec{t}]\!] |\!\})$. Hence, the result. ∎

Theorem 4 *Let $conf = \langle C, \rho\rangle$ be an initial configuration. Let $conf_1 \ldots conf_m$ be such that every non-failing computation of $conf$ passes through a $conf_i$. Then, $\mathcal{M}[\![conf]\!]env = \biguplus_i \mathcal{M}[\![conf_i]\!]$.*

Proof: The proof follows by induction on the number of steps of reduction, if we prove the result for one step reductions. Thus we need to prove the following: if

$conf_1 \ldots conf_m$ be all the possible configurations attainable in one step from $conf$, then $\mathcal{M}[\![conf]\!]env = \uplus_i \mathcal{M}[\![conf_i]\!]$. This is immediate from the previous two lemmas and the compositionality of the denotational semantics. ∎

4.4.2 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 [40].

The rest of this section is organized as follows. As in the proof in Chapter 2, we first develop notation to relate syntactic and semantic values. Next, the proof is done for the determinate case. This proof resembles the adequacy proof in Chapter 2. Finally, we use the properties of the powerdomain to extend the proof for the determinate case to the full calculus.

Operational properties

We first define a transition relation that is useful in the proof. Intuitively, the relation $\longmapsto_s$ differs from $\longrightarrow$ in allowing addition of new atoms to the operational configuration.

Definition 54 *(Definition of relation $\longmapsto_s$)*

- $conf_1 \longrightarrow conf_2 \Rightarrow conf_1 \longmapsto_s conf_2$
- $\langle C, \rho\rangle \longmapsto_s \langle C, C', \rho\rangle$

When $\langle C, \rho\rangle \overset{*}{\longmapsto}_s \langle D, \rho'\rangle$, we will sometimes write $\langle C, \rho\rangle \overset{*}{\longmapsto}_s \langle U_C \uplus U_e, \rho'\rangle$, to indicate that the atoms in U_C arose by reductions from C. Thus the atoms in U_e arise from the atoms added in step two of the above definition.

Lemma 77 $(\langle C, \rho\rangle \longmapsto_s \langle C, \rho'\rangle) \wedge env \preceq \rho \Rightarrow env \preceq \rho'$

The transition systems $\longrightarrow$ and $\longmapsto_s$ are closely related. The following lemma is a Church-Rosser like property. It can be viewed as saying that addition of new atoms cannot disable enabled reductions.

Lemma 78 *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 79 *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* $\langle C,\rho\rangle \stackrel{*}{\longmapsto}_s conf_1 \ \wedge \ \langle C,\rho\rangle \stackrel{*}{\longmapsto}_s conf_2$. *Then, there is a configuration* $conf$ *such that* $conf_1 \stackrel{*}{\longmapsto}_s conf \ \wedge \ conf_2 \stackrel{*}{\longmapsto}_s conf$.

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 environment was defined. The definition is reproduced 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. $\dfrac{< e, \rho > \longrightarrow t}{< e \uparrow s, \rho > \longrightarrow t \uparrow s}$

Lemma 80 *Let ρ be a consistent syntactic environment. Let $< e, \rho > \longrightarrow f_n^i$, where f_n^i is a function symbol. Then, $< e, \rho > \longrightarrow f_n^i$ is independent of the choice made in rule 2 of the transition system above.*

The following definition is intended to set a up a relationship between syntactic and semantic objects. The first case of the 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. We write $\vec{a} \preceq (\vec{t}, \rho)$ as shorthand for $a_i \preceq (t_i, \rho), i = 1 \ldots n$. The definition is then extended to relate syntactic and semantic environments. This can be viewed as saying that the syntactic environment ρ is more constrained than env. ρ dominates env, written $env \preceq \rho$. The third case of the definition combines the first two in a natural manner.

Definition 55 *Definition of $\preceq$:*

1. *Relating terms and semantic values:*
   - $v \preceq (x, \rho)$ *is true if ρ consistent implies that for all finite sequences s,* $v[s] \in W_f \Rightarrow \langle x \uparrow [s|0], \rho \rangle \xrightarrow{*} f$
   - $v_1 \preceq (t_1, \rho) \wedge \ldots v_n \preceq (t_n, \rho) \Rightarrow f(v_1 \ldots v_n) \preceq (f(t_1, \ldots t_n), \rho)$
2. *Relating semantic and syntactic environments:*
   $env \preceq \rho \Leftrightarrow \rho$ *inconsistent* $\vee\ env \neq env_\top \wedge (\forall x)\ [env(x) \preceq (x, \rho)]$
3. $(env, a) \preceq (\rho, t) \Leftrightarrow a \preceq (t, \rho) \wedge env \preceq \rho$

Both the relations defined above satisfy expected monotonicity properties.

Lemma 81 *(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*([66]). The proof is immediate and is omitted.

Lemma 82 *(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* $[(\forall env_i)\ env_i \preceq \rho] \Rightarrow env \preceq \rho$

Determinate case

This section is essentially an adaptation of the adequacy proof of Chapter 2 to the present setting. Recall that the proof in Chapter 2 worked only for determinate programs. In this section, we consider logic programs in which predicate names have only one definition, thus making them determinate. The next two sections shows how to construct the adequacy proof for the full calculus, using the proof done in this section as building block.

As in Chapter 2, we define inclusive predicates relating syntactic and semantic entities.

Definition 56 *(Definition of inclusive predicate for terms)*
Let $g \in [ENV \times {}^{\top}\Pi_n^{\top} V \xrightarrow{c} ENV \times {}^{\top}\Pi_n^{\top} V \times \mathbf{2}]$. $\vec{t} \preceq g$ *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 operational semantics after a finite number of steps. In particular, the semantics handles failed computations too. In the following definition, we use $\star$ in the environment part to indicate that the actual environment is not relevant to the definition.

Definition 57 *(Definition of inclusive predicate)*
Let $g \in [ENV \xrightarrow{c} ENV \times \mathbf{2}]$. $g \preceq C$ *if the following holds. Let,*

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

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

- $\langle U_C \uplus U_e, \rho \rangle \xrightarrow{*} \langle U'_C \uplus U'_e, \rho' \rangle$
- $env_f \preceq \rho'$
- $term = \top \Rightarrow U'_C = \phi$

Suppose that $g_1, g_2 \in [ENV \xrightarrow{c} ENV \times 2]$ 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. This is the analogue of the Interleaving lemma of Chapter 2.

Lemma 83 *Let* $g, h \in [ENV \xrightarrow{c} ENV \times 2]$. *Then,*

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

Proof: Let

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

Let $\langle env_i, term_i \rangle = (g \circ h)^i\ env$. We prove by induction on i that $(\forall env_f \sqsubseteq env_i)$, there is a configuration $\langle U'_C \uplus U'_D \uplus U_e, \rho_{res} \rangle)$ such that

- $\langle U_C \uplus U_D \uplus U_e, \rho \rangle \xrightarrow{*} \langle U'_C \uplus U'_D \uplus U_e, \rho_{res} \rangle$
- $env_f \preceq \rho_{res}$.
- $term_i = \top \Rightarrow [U'_C \uplus U'_D = \phi]$

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 $\langle U_D, \rho\rangle \stackrel{*}{\longmapsto}_s \langle U''_C \uplus U''_D \uplus U_e, \rho_2\rangle$ and $h \preceq D$, $(\exists\langle U''_C \uplus U'_D \uplus U_e, \rho_2\rangle)$ such that

- $\langle U''_C \uplus U''_D \uplus U_e, \rho_2\rangle \xrightarrow{*} \langle U''_C \uplus U'_D \uplus U_e, \rho_1\rangle$
- $env_1 \preceq \rho_1$
- $term_i = \top \Rightarrow U'_D = \phi$

Since $\langle U_C, \rho\rangle \stackrel{*}{\longmapsto}_s \langle U''_C \uplus U'_D \uplus U_e, \rho_1\rangle$ and $g \preceq C$, $(\exists\langle U'_C \uplus U'_D \uplus U_e, \rho_{res}\rangle)$ such that

- $\langle U''_C \uplus U'_D \uplus U_e, \rho_1\rangle \xrightarrow{*} \langle U'_C \uplus U'_D \uplus U_e, \rho_{res}\rangle$
- $env_f \preceq \rho_{res}$

- $term_i = \Rightarrow U'_C = \phi$

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 84 $g \preceq C \Rightarrow \mathcal{E}[\![G]\!] | g \preceq G | C$

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, $(\mathcal{E}[\![G]\!] | g)\ env = env_\top$. Operationally, guard G evaluates to false and the execution of $G|C$ in ρ fails. ∎

The meanings of predicate symbols are elements of $[\Pi_n^\top V \xrightarrow{c} \Pi_n^\top V \times 2]$. So, we need to develop some notation to relate elements of $[\Pi_n^\top V \xrightarrow{c} \Pi_n^\top V \times 2]$ and predicate symbols. The motivations for these definitions are quite the same as before. As before, we use $\star$ in the environment part to indicate that the actual environment is not relevant to the definition.

Definition 58 *(Definition of inclusive predicate)*
Let $h \in [\Pi_n^\top V \xrightarrow{c} \Pi_n^\top V \times 2]$. $h \preceq p$ *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*

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

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

Lemma 85 *(Renaming lemma)*
Let $\lambda : Y_1 \to Y_2$ *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 86 $g \preceq \{G|C\} \Rightarrow h \preceq p$ *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

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

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

$$\langle p(\vec{t}), \star\rangle \longrightarrow \langle G^x_y | C^x_y, \star\rangle$$

where $G^x_y|C^x_y$ is supposed to indicate a freshly renamed clause with variables $\vec{y}$ for $\vec{x}$. From 79, we deduce the existence of a configuration $conf$ such that

$$\langle G^x_y | C^x_y, \star\rangle \overset{*}{\longmapsto}_s conf \wedge \langle U, \rho\rangle \overset{*}{\longrightarrow} conf$$

From lemma 85, $g^x_y \preceq G^x_y|C^x_y$. 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 87

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

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

Proof: Note that we have,

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

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

Full proof

In this section, we extend the proof for the determinate case to the full calculus. This is done in two stages. First, we define a notion of determinisation of programs. Using these ideas, we develop inclusive predicates to relate elements of the powerdomain to syntactic entities. Finally, the full proof is presented.

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 be 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:

$$N^\omega \subseteq R$$

$$(\forall n)\ [s_1 \ldots s_n \in R \ \Rightarrow \ \langle s_1 \ldots s_n\rangle \in R]$$

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

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

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

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

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 $q_1(.), \ldots q_n(.)$ is an n-tuple $\langle c_1 \ldots c_n \rangle$, 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.

Inclusive predicates

This section defines the inclusive predicates for relating elements of the powerdomain to indeterminate programs. The intuitive idea is to lift the the "dominates" relation from determinate programs to indeterminate programs.

The following definition relates elements of $\overline{P([\Pi_n^\top V \xrightarrow{c} \Pi_n^\top V \times 2])}$, say H to predicate symbols p. Recall that we have defined a "dominates" relation between determinate programs and elements of $[V \xrightarrow{c} V \times 2]$. Informally, the following definition can be viewed as follows: Pick any determinisation d of the program. Then, there is an element in the "set" of functions H such that d "dominates" H.

Definition 59 *Let $H \in \overline{P([\Pi_n^\top V \xrightarrow{c} \Pi_n^\top V \times 2])}$. Then, $H \preceq p$ if*

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

The following definition relates ground atoms of the form $p(\vec{t})$ to the elements of $\overline{P([ENV \times^\top \Pi_n^\top V \xrightarrow{c} ENV \times^\top \Pi_n^\top V \times 2])}$ to .

Definition 60 *If $F \in \overline{P([ENV \times^\top \Pi_n^\top V \xrightarrow{c} ENV \times^\top \Pi_n^\top V \times 2])}$, define $F \preceq p(\vec{t})$ if $(\forall p^d \in Det(p))\ (\exists f)\ [F \uplus \{\!| f |\!\} = F \wedge f \preceq p_d(\vec{t})]$.*

The following definition relates elements of $\overline{P([ENV \xrightarrow{c} ENV \times 2])}$, to sequences of clauses C.

Definition 61 *Let $G \in \overline{P([ENV \xrightarrow{c} ENV \times 2])}$. Then, $G \preceq C$ if*

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

It is easy to check that all the definitions above define inclusive predicates.

Full proof

This section is the heart of the adequacy proof. Recall that the inclusive predicates were defined by lifting the "dominates" relation from determinate programs to indeterminate programs. The proof below has the same structure. It reduces the cases of structural induction to the corresponding cases of the structural induction for the determinate case.

Lemma 88 *(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* $p(\vec{u_1}) \leftarrow Gu_1|C_1 \ldots p(\vec{u_n}) \leftarrow Gu_n|C_n$. *Define new predicate names* p_i, *for* $i = 1 \ldots n$, *by the unique clause* $p_i(\vec{u_i}) \leftarrow Gu_i|C_i$. *Then,*

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

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

Proof:

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 83, 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 84 $\mathcal{E}[\![Gu|g]\!] \preceq Gu|c$. 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, $H \uplus h_i = H \wedge h_i \preceq p^d$.

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 86, $h \preceq p^d$ where

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

Thus,

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

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 $h_g = Id_{ENV} \times^\top h || g$. From lemma 87, $h_g \preceq p^d$. Furthermore,

$$\begin{aligned} H_g \uplus \{\!|\ h_g\ |\!\} &= (Id_{ENV} \times^\top H) || \{\!|\ g\ |\!\} \uplus \{\!|\ (Id_{ENV} \times^\top h) || g\ |\!\} \\ &= (Id_{ENV} \times^\top H) || \{\!|\ g\ |\!\} \uplus (Id_{ENV} \times^\top \{\!|\ h\ |\!\}) || \{\!|\ g\ |\!\} \\ &= (Id_{ENV} \times^\top (H \uplus \{\!|\ h\ |\!\})) || \{\!|\ g\ |\!\} \\ &= (Id_{ENV} \times^\top H) || \{\!|\ g\ |\!\} \\ &= H_g \end{aligned}$$

Thus, we have proved all cases of the lemma. ∎

Theorem 5 $\mathcal{E}[\![p]\!] \preceq p$

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 88. We give the proof for the recursive case below.

Let the defining clauses for p be given by

$$p(\vec{x}) \leftarrow Gu_1 | C_1$$

$$\vdots$$

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

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 88 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 the relevant cases of lemma 88.

- The ith defining clause for p contains references to p. Call this p_i. Proof is by using the relevant cases of lemma 88, 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 predicate $\preceq$, $\mathcal{E}[\![p]\!] \preceq p$. ∎

4.4.3 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 *Primobs* into finite elements of ENV. This is straightforward: $t_1 \sqsubseteq \rho(x_1)$ corresponds to $env_\perp[x_1 \mapsto t]$. Note that it is straightforward to extend this to a conjunction of primitive observations: $t_1 \sqsubseteq \rho(x_1) \wedge \ldots t_n \sqsubseteq \rho(x_n))$ maps to $env_\perp[x_1 \mapsto t, \ldots x_n \mapsto t_n]$. This can be extended to a translation function $Trans$ mapping elements of OBS to finite elements of $\overline{P_S(ENV \times \mathbb{2})}$. For the purposes of this discussion, assume that the elements of OBS are in disjunctive normal form, i.e as disjunctions of conjunctions. The first two cases of the following definition is the base case: they describe the translation for a conjunct. The last case describes how to construct the translation of a disjunction: this uses the union operation of the powerdomain.

1. $Trans\langle t_1 \sqsubseteq \rho(x_1) \wedge \ldots t_n \sqsubseteq \rho(x_n), donotcare\rangle = \langle env_\perp[x_i \mapsto t_i, i = 1 \ldots n], \perp\rangle$

2. $Trans\langle t_1 \sqsubseteq \rho(x_1) \wedge \ldots t_n \sqsubseteq \rho(x_n), terminated\rangle = \langle env_\perp[x_i \mapsto t_i, i = 1 \ldots n], \top\rangle$

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

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

Lemma 89 $\langle p(\vec{t}), \phi\rangle \vdash s \Leftrightarrow Trans(s) \sqsubseteq \mathcal{E}[\![p(\vec{t})]\!]\langle env_\perp, \perp\rangle$

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 5, 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 57 of $g \preceq p^d$, 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, the computation tree is finitely branching. Thus, 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 4, $\mathcal{E}[\![\langle p(\vec{t}), \phi\rangle]\!]env_\perp = \biguplus \mathcal{E}[\![conf_i]\!]env_\perp$. Hence the result. ∎

Theorem 6 *(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

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

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 $Test_s$. However, $Trans(Test_s) = s$ is not less than $\mathcal{E}[\![q(t_x)]\!]$. Thus, $\langle p(t_x), \phi\rangle$ does not pass $Test_s$. ∎

4.5 Semantics of full language

This section extends the semantics to the full language. The syntactic differences arise from the more powerful tests allowed in the guard 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 $x = t$. A syntactic environment ρ models a primitive 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 algebraic 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 goes through essentially unchanged.

The semantic domain of environments is constructed from the syntactic environments. Intuitively, 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 ordered by the "refinement ordering". A more explicit description of the domain ENV is given below.

Definition 62 *ENV is the preorder defined as follows.*

- *The elements of ENV are (possibly) infinite syntactic environments, 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 90 *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 4.2.1.

The space of values V is defined as follows.

Definition 63 *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 91 *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 $\overline{P([V^n \xrightarrow{c} V^n \times 2])}$. 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 alias 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 available for use as $\vec{x}$ in the above definition. Furthermore, note that the result returned is *independent* of the choice of the variable names $\vec{x}$ as long as they satisfy the disjointness condition.

4.6 Conclusions

This chapter describes an investigation into an abstract semantics for a concurrent logic programming language. The operational semantics was presented in a Plotkin-style structured operational semantics. The abstract semantics showed that programs could be given a simple denotational treatment couched entirely in terms of equations and equation solving. The denotational semantics was proved to be fully abstract with respect to the operational semantics. 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.

The language considered here was based on the constraint system built out of Herbrandt terms. The semantic techniques developed here can be applied to any language in which objects are created through constraint intersection: for example, languages over general constraint systems [57].

There is one major unsatisfying feature of the semantics discussed in this chapter. The semantics described in this the treatment of error in computations as "benign": for example, the following two programs p and q are identified: p has one definition

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

q has two definitions as follows:

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

In the framework of committed choice languages, these two programs should not be identified. The latter program has a computation path that terminates in $error$ whereas the former program does not. In a committed choice language, the computation path that ends in $error$ is "observable": this is the meaning of committed choice.

It should be noted that previous work [17,57,13] handles the problem outlined above: thus, $error$ is made "observable". However, the view of programs that was presented in this chapter is far less detailed than the view of programs in these papers: in particular we do not keep track of computation paths, and describe parallel composition abstractly as constraint intersection, without explicit operational interleaving of the conjuncts. In future work, we hope to extend this "simpler" view of programs to include a proper treatment of *error*.

Bibliography

[1] S. Abramsky. Domains in logical form. In *Proceedings of the Symposium on Logic in Computer Science*, pages 47–53, 1987.

[2] S. Abramsky. Unpublished lecture at mfps, 1989.

[3] S. Abramsky. Computational interpretations of linear logic. Technical Report Research Report DOC 90/20, Imperial College, 1990.

[4] S. Abramsky and C. H. L. Ong. Full abstraction in the lazy lambda calculus. *Submitted to Information and Computation*, 1988.

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

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

[7] G. Berry. Some syntactic and categorical constructions of lambda-calculus models. Technical Report 80, Institute National de Recherche en Informatique et en Automatique (INRIA), 1981.

[8] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, 1990.

[9] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *Information and Computation*, 1990. To appear.

[10] G. Boudol. Towards a lambda-calculus for concurrent and communicating systems. In J. Diaz, editor, *TAPSOFT 89, Lecture Notes in Computer Science 351*, pages 149–161. Springer-Verlag, 1989.

[11] J. D. Brock and W. B. Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts*, pages 252–259, 1981. LNCS 107.

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

[13] F. S. de Boer and C. Palamidessi. On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences. In *Proceedings of CONCUR 90, Lecture notes in Computer Science, 458*, pages 99–114, 1990.

[14] R. deNicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1983.

[15] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

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

[17] R. Gerth, M. Codish, Y. Lichtenstein, 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] C. Gunter. Relating total and partial correctness interpretations of non-deterministic programs. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 306–319, 1990.

[19] C. Gunter. The mixed powerdomain. *Theoretical Computer Science*, 1991. In print.

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

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

[22] M. Hennessy and G. Plotkin. A term model for ccs. In *Proceedings of Mathematical Foundations of Computer Science*, 1980.

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

[24] C. A. R. Hoare, S. Brookes, and W. Roscoe. A model for communicating sequential processes. Technical report, Programming Research Group, Oxford, 1981.

[25] J. Jaffar, J. L. Lassez, 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.

[26] 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.

[27] 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. To appear in ACM Transactions on Programming Langauges and Systems.

[28] R. Jagadeesan and K. Pingali. Abstract semantics for a higher-order functional language with logic varaibles. Technical Report TR 91-1120, Cornell, 1991.

[29] R. Jagadeesan and V. Shanbhogue. Closure operator semantics for concurrent constraint logic programming. Technical Report TR 90-1174, Computer Science Department, Cornell University, December 1990.

[30] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Sciences Laboratory, Xerox PARC, January 1991.

[31] B. Jonsson. A fully abstract trace model for dataflow networks. In *Proceedings of the Sixteenth Annual ACM Symposium On Principles Of Programming Languages*, 1989.

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

[33] G. Kahn and D. MacQueen. Coroutines and networks of parallel proceses. In *Proceedings of IFIP Congress*. North-Holland, 1977.

[34] J. Kok. A fully abstract semantics for dataflow nets. In *Proceedings of Parallel Architectures And Languages Europe 1987*, pages 351–368, Berlin, 1987. Springer-Verlag.

[35] 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.

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

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

[38] N. A. Lynch and E. W. Stark. A proof of the kahn principle for input/output automata. *Information and Computation*, 1989.

[39] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill Publishing Company, 1981.

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

[41] R.E. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London and John Wiley, New York, 1976.

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

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

[44] R. Milner. A modal characterisaiton of observable machine behaviour. In E. Astesiano and C. Bohm, editors, *CAAP'81: Trees in algebra and Programming, volume 112, Lecture notes in Computer Science*, pages 25–34, 1981.

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

[46] J. Misra. Equational reasoning about nondeterministic processes. unpublished manuscript, 1989.

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

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

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

[50] C. H. L. Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.* Ph.D. dissertation, Imperial College of Science and Technology, 1988.

[51] G. D. Plotkin. Lecture notes on domain theory. The Pisa Notes.

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

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

[54] Gordon Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[55] A. Rabinovich and B. A. Trakhtenbrot. Nets and data flow interpreters. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 164–174, 1989.

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

[57] 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. To appear.

[58] J. R. Russell. Full abstraction for nondeterministic dataflow networks. In *Proceedings of the 30th Annual Symposium of Foundations of Computer Science, Lecture Notes in Computer Science 442*, pages 170–177, 1989.

[59] J. R. Russell. On oraclizable networks and kahn's principle. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1990.

[60] V. A. Saraswat. *Concurrent Constraint Programming Languages.* Ph.D. dissertation, Carnegie-Mellon University, 1988.

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

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

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

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

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

[66] 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.

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

[68] C. P. Wadsworth. The relation between computational and denotational properties for scott's d_∞ models of the λ-calculus. *SIAM J. Computing*, 5:488–521, 76.

[69] D. J. Walker. Bisimulation and divergence. In *Third IEEE Symposium on Logic in Computer Science*, pages 186–192, 1988.

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

Appendix A

Proof of Strong Normalisation of labelled calculus

The terms of the labelled calculus with bottom, denoted $BC^{\omega}\bot$ is defined by the folowing grammar

Terms $::= x \mid\mid \lambda\langle x_1 \ldots x_k\rangle.M \mid\mid MN \mid\mid M|N \mid\mid M^n$

where $n \in \omega$.

Definition 64 *The syntactic equality $\equiv$ is the congruence (with respect to substitution) that is generated by the equation:*$p|(q|r) \equiv (p|q)|r$

We need basic notation to be able to refer to redices. This is done in a style similar to the techniques used in the study of the operational semantics of the unlabelled calculus. Most of the definitions that appear below have been used before in the paper, but are repeated here to make this section self contained.

Define, by mutual recursion:

$Terms_1 ::= x \mid\mid \lambda\langle x_1 \ldots x_k\rangle.M \mid\mid MN \mid\mid M^n$

$Terms_2 ::= M \mid\mid M|N$

where $M,\ N \in Terms_1 \bigcup Terms_2$. The intuitive meanings of these classes are the same as the corresponding classes in the unlabelled calculus.

Intuitively, $Terms_2$ are the terms of the form $t_1|\ldots|t_n$, where the t_i are either abstractions or applications. The following definition is intended to capture the "number" of $t_i's$.

Define $len : Terms_2 \rightarrow Int$ as follows:

- $len(p) = 1$, if $p \in Terms_1$
- $len(p|q) = len(p) + len(q)$

It can be checked that this function is well-defined on the terms quotiented by the syntactic equality $\equiv$. The following definition is intended to capture the "position" of t_i in $t_1|\dots t_n$. Define a partial function $index : \omega \times Terms_2 \rightarrow Terms_1$ as follows:

- $index(n, p) =$ undefined if $len(p) \leq n \wedge\ len(p) \neq n$
- $index(1, p) = p$, if $p \in Terms_1$
- $index(n, p|q) = index(n, p)$, if $n \leq len(p)$
- $index(n, p|q) = index(n - len(p), q)$, if $len(p) \leq n \wedge len(p) \neq n$

Let $[x \mapsto N]M$ be notation for the usual notion of substitution. Let $M \in Terms_2 \wedge 1 \leq s \leq len(M)$. Then, $M[s \mapsto N]$ is notation for the term $M' \in Terms_2$, such that

- $len(M') = len(M)$
- If, $1 \leq t \leq len(M') \wedge s \neq t$, $index(t', M) = index(M, t)$
- $index(s, M') = N$

The reduction relation is presented as a transition system. We also formalise the notion of "different" reductions. This is done by associating an ordered pair with a reduction, that captures the position and argument of the abstraction that is involved in the reduction.

- $(M^m)^n \rightarrow_{l\langle 1,0\rangle} M^{min(m,n)}$
- $(M^0) \rightarrow_{l\langle 1,0\rangle} \perp$
- $\perp^n \rightarrow_{l\langle 1,0\rangle} \perp$
- $\perp^n N \rightarrow_{l\langle 1,0\rangle} \perp$
- $(\lambda\langle x_1 \dots x_k\rangle.M)N \rightarrow_{l\langle 1,i\rangle} \lambda\langle x_1 \dots x_{i-1}, x_{i+1} \dots x_k\rangle.[x_i \mapsto N]M$
  if $1 \leq i \leq k$

- If $P \in Terms_2 \wedge\ 1 \leq s \leq len(P)$, then
  $PN \rightarrow_{l\langle s,0\rangle} M[s \mapsto index(s,P)N]$

- $M \rightarrow_{l_\sigma} M' \Rightarrow M|N \rightarrow_{l_\sigma} M'|N$

- $N \rightarrow_{l_\sigma} N' \Rightarrow M|N \rightarrow_{l_{\sigma'}} M|N'$
  where $\sigma' = \langle first(\sigma) + len(M), second(\sigma)\rangle$

- $M \rightarrow_{l_\sigma} M' \Rightarrow MN \rightarrow_{l\langle 1,\sigma\rangle} M'N$

- $N \rightarrow_{l_\sigma} N' \Rightarrow MN \rightarrow_{l\langle 2,\sigma\rangle} MN'$

- $M \rightarrow_{l_\sigma} M' \Rightarrow M^n \rightarrow_{l\langle 1,\sigma\rangle} M'^n$

Two reductions $M \rightarrow_{\sigma'} M'$, and $M \rightarrow_{\sigma''} M''$ are different if $\sigma' \neq \sigma''$. Note that there are only finitely many different reductions. $\xrightarrow{*}_l$ is the reflexive and transitive closure of $\rightarrow_l$.

Let $M \in \Lambda_0$. Construct a tree with labelled edges corresponding to M denoted by $T(M)$ as follows. Let $M \rightarrow_{\sigma_i} M_i$, be all the possible different one step reductions from M. Then, the root has an edge for each label σ_i. The subtree at the node at the other end of the edge with label σ_i is the one obtained by doing the construction for M_i. Define the set of strongly normalising terms in $BC^\omega\bot$ as follows.
$SN = \{M|$there is no infinite reduction sequence $M = M_0 \rightarrow M_1 \rightarrow M_2 \ldots\}$. By a Konig's lemma argument, we deduce that M is SN $\Rightarrow T(M)$ is finite.

Definition 65 *Let $\langle D, \leq\rangle$ be the domain of labelled, finitely-branching trees of finite depth, where the ordering relation $\leq$ is the subtree ordering.*

Note that $\langle D, \leq\rangle$ is well-founded. Furthermore, if $M \in SN$ and $M \rightarrow M'$, then $T(M') \leq T(M)$, $T(M') \neq T(M)$. $T(M)$ plays a role analogous to the role of $d(M)$ which is the length of the reduction sequence from M in the proof of strong normalisation of the labelled lambda calculus, as in Barendregt's book [6].

We need to keep track of redices explicitly. First,we define positions of subterms in a term. We define a partial function $F_M : Seq^\star \rightarrow Terms$, as follows. The intuition is that $F_M(\sigma)$, is the subterm of M at position σ. This is done by structural induction.

- $M \in Terms_1$

- $M \in Var$. Say, $M = z$. Then, $F_z(\langle 0,0\rangle) = z$
- $M = PN$. The,
  - $F_M(\langle 0,0\rangle) = M$
  - $F_M(\langle 1,\sigma\rangle) = F_P(\sigma)$
  - $F_M(\langle 2,\sigma\rangle) = F_N(\sigma)$
- $M = \lambda\langle x_1 \ldots x_k\rangle.P$. Let σ be of form $\langle,\rangle$.
  - $F_M(\langle 0,0\rangle) = M$
  - $F_M(\langle 1,\sigma\rangle) = F_P(\sigma)$
- $M = P^n$. Let σ be of form $\langle,\rangle$.
  - $F_M(\langle 0,0\rangle) = M$
  - $F_M(\langle 1,\sigma\rangle) = F_P(\sigma)$

- $M \in Terms_2 \wedge\ len(M) = n+1 \wedge\ 1 \leq n$. Let $1 \leq i \leq len(M)$. Then
  - $F_M(\langle 0,0\rangle) = M$
  - $F_M(\langle i,\sigma\rangle) = F_{index(i,M)}(\langle 1,\sigma\rangle), 1 \leq i$

Let $M \in BC^\omega\bot$. Let $mark : BC^\omega\bot \times Seq^\star$ be a predicate. The intuition is that certain subterms of M are marked out. The following definition passes marking information through reductions. This is done by analysing the structure of the reductions.

Definition 66 *(Extending marking through reductions)*

- $(M^m)^n \rightarrow_{l\langle 1,0\rangle} M^{min(m,n)}$. *Then*
  - $mark((M^m)^n, \langle 1,\langle 1,\sigma\rangle\rangle) \Rightarrow mark(M^{min(m,n)}, \langle 1,\sigma\rangle)$
  - $mark((M^m)^n, \langle 1,\langle 0,0\rangle\rangle)$ *or* $mark((M^m)^n, \langle 1,\langle 1,\langle 0,0\rangle\rangle\rangle)$ *implies* $mark(M^{min(m,n)}, \langle 0,0\rangle)$.
- $(M^0) \rightarrow_{l\langle 1,0\rangle} \bot$: $mark(M^0, \langle 0,0\rangle) \vee\ mark(M^0, \langle 1,\langle 0,0\rangle\rangle) \Rightarrow\ mark(\bot, \langle 0,0\rangle)$.
- $\bot^n \rightarrow_{l\langle 1,0\rangle} \bot$: $mark(\bot^n, \langle 0,0\rangle) \vee mark(\bot^n, \langle 1,\langle 0,0\rangle\rangle) \Rightarrow\ mark(\bot, \langle 0,0\rangle)$.
- $\bot N \rightarrow_{l\langle 1,0\rangle} \bot$. *Then* $mark(\bot N,\ \langle 1,\langle 0,0\rangle\rangle)$ *implies* $mark(\bot, \langle 0,0\rangle)$.

- $(\lambda\langle x_1 \dots x_k\rangle.M)N \rightarrow_{l\langle 1,\, i\rangle} \lambda\langle x_1 \dots x_{i-1}, x_{i+1} \dots x_k\rangle.[x_i \mapsto N]M \wedge\ 1 \leq i \leq k$. *Let* $P = \lambda\langle x_1 \dots x_{i-1}, x_{i+1} \dots x_k\rangle.[x_i \mapsto N]M$. *Then, if* $F_P(\sigma)$ *is defined* $mark((\lambda\langle x_1 \dots x_k\rangle.M)N, \langle 1, \langle 0,0\rangle\rangle) \Rightarrow mark(P, \sigma)$
- $(\lambda\langle x_1 \dots x_k\rangle.M)^{(n+1)}N \rightarrow_{l\langle 1,\, i\rangle} \lambda\langle x_1 \dots x_{i-1}, x_{i+1} \dots x_k\rangle.[x_i \mapsto N^n]M \wedge 1 \leq i \leq k$. *Let* $P = \lambda\langle x_1 \dots x_{i-1}, x_{i+1} \dots x_k\rangle.[x_i \mapsto N^n]M$. *Let* $F_P(\sigma)$ *be defined. Then,* $mark(P, \sigma)$, *if* $mark((\lambda\langle x_1 \dots x_k\rangle.M)^{(n+1)}N, \langle 1, \langle 0,0\rangle\rangle)$ *holds* *or* $mark((\lambda\langle x_1 \dots x_k\rangle.M)^{(n+1)}N, \langle 1, \langle 1, \langle 0,0\rangle\rangle\rangle)$ *holds.*

Marking information is extended to the whole term by structural induction.

- $M = NP$. *Then,*
  - $mark(N, \sigma) \Rightarrow mark(MN, \langle 1, \sigma\rangle)$
  - $mark(P, \sigma) \Rightarrow mark(MN, \langle 2, \sigma\rangle)$
- $M = M_1|M_2$. *Let* $len(M) = s,\ 1 \leq i \leq s$. *Then,* $mark(index(i, M), \sigma) \Rightarrow mark(M, \langle i, \sigma\rangle)$

Lemma 92 *Let* $M \in Terms_2 \wedge\ len(M) = k \wedge\ M \xrightarrow{*}_l P$. *Then*

- $P \in Terms_2$
- *Let* $len(P) = s$. *Then,*
  $(\exists i_1 \dots i_{k+1})\ [(1 \leq i_1 \leq i_2 \dots \leq i_k \leq i_{k+1} = s+1) \wedge (i_1 \neq i_2 \wedge\ \dots i_{k-1} \neq i_k \wedge\ i_k \neq$
  i_{k+1}
  such that
  $(\exists P_1 \dots P_k)\ (\forall j = 1 \dots k)\ [index(j, M) \xrightarrow{*}_l P_j \wedge\ P_j \in Terms_2]$, *and*
  - $(\forall j = 1 \dots k)\ [len(P_j) = i_{j+1} - i_j]$
  - $(\forall j = 1 \dots k)\ (\forall t = 1 \dots len(P_j))\ [index(t, P_j) = index(i_j + t - 1, P)]$

Proof: Induction on the length of reduction $M \xrightarrow{*}_l P$. ∎

Let $N \in BC^{\omega}\bot$. Then, $M^\star$ is notation for $[x \mapsto N]M$.

Lemma 93 *Let* $M \in Terms_2 \wedge\ len(M) = k$. *Then*

- $M^\star \in Terms_2$

- *Let* $len(M) = k$, $len(M^\star) = s$. *Then,*
  $(\exists i_1 \ldots i_{k+1})\ [(1 \leq i_1 \leq i_2 \ldots \leq i_k \leq i_{k+1} = s+1) \wedge (i_1 \neq i_2 \wedge\ \ldots i_{k-1} \neq i_k \wedge i_k \neq i_{k+1})$
  such that
  $(\exists P_1 \ldots P_k)\ (\forall j = 1 \ldots k)\ [(index(j, M))^\star = P_j \wedge\ P_j \in Terms_2]$, *and*
  - $(\forall j = 1 \ldots k)\ [len(P_j) = i_{j+1} - i_j]$
  - $(\forall j = 1 \ldots k)\ (\forall t = 1 \ldots len(P_j))\ [index(t, P_j) = index(i_j + t - 1, M^\star)]$

Proof: Structural induction. ∎

Lemma 94 *Let* $M \in Terms_2 \wedge\ len(M) = k \wedge M^\star \xrightarrow{*}_l P$. *Then*

- $P \in Terms_2$
- *Let* $len(P) = s$. *Then,*
  $(\exists i_1 \ldots i_{k+1})\ [(1 \leq i_1 \leq i_2 \ldots \leq i_k \leq i_{k+1} = s+1) \wedge (i_1 \neq i_2 \wedge\ \ldots i_{k-1} \neq i_k \wedge i_k \neq i_{k+1})$
  such that
  $(\exists P_1 \ldots P_k)\ (\forall j = 1 \ldots k)\ [(index(j, M))^\star \xrightarrow{*}_l P_j \wedge\ P_j \in Terms_2]$, *and*
  - $(\forall j = 1 \ldots k)\ [len(P_j) = i_{j+1} - i_j]$
  - $(\forall j = 1 \ldots k)\ (\forall t = 1 \ldots len(P_j))\ [index(t, P_j) = index(i_j + t - 1, M^\star)]$

Proof: From lemma 92 and lemma 93. ∎

Let $M \in Terms_2 \wedge\ len(M) = k \wedge M^\star \xrightarrow{*}_l P$. From above lemma, under the hypothesis of the above lemma, there is a well defined function
$f : \{1 \ldots len(P)\} \rightarrow len(M)$ such that

- $index(f(s), M) \xrightarrow{*}_l P_{f(s)} \wedge (\exists 1 \leq t \leq len(P_{f(s)}))[index(t, P_{f(s)}) = index(s, P)]$
- $i_{f(s)} \leq s \leq (i_{f(s)} + len(P_{f(s)}) - 1)$

The following special class of terms plays an important role in the proof of the strong normalisation theorem. Define, by mutual recursion:
$T_1 ::= x \mid\mid T_2 N \mid\mid M^n$
$T_2 ::= T_2|P \mid\mid P|T_2 \mid\mid T_1$
where $P \in BC^\omega\bot$

Consider $M^\star = [x \mapsto N]M$. Let $M^\star \rightarrow_l P$. For the proofs, we need to trace the terms of P that arise from the N's that have been substituted for x. This is done using the marking machinery that has been set up earlier. The initial marking on $M^\star$ is defined by structural induction. Without loss of generality, assume that x does not occur as the variable bound in an abstraction.

- $R = [x \mapsto N]x$. Then,
  $F_R(\sigma)$ defined $\Rightarrow$ $mark(R, \sigma)$
- $R = [x \mapsto N](R_1 R_2)$. Then,
  - $mark([x \mapsto N]R_1, \sigma) \Rightarrow mark([x \mapsto N](R_1 R_2), \langle 1, \sigma\rangle)$
  - $mark([x \mapsto N]R_2, \sigma) \Rightarrow mark([x \mapsto N](R_1 R_2), \langle 2, \sigma\rangle)$
- $M = [x \mapsto N](M_1 | M_2)$. Let $len(M) = s$, $1 \leq t \leq s$. Let $i_j \leq t \leq i_{j+1} \wedge\ t \neq i_{j+1}$, where the i_j's are the indices given by lemma 93. Then,
  - $i_j \leq len(M_1) \wedge mark([x \mapsto N]index(t, M_1), \sigma) \Rightarrow mark(M, \langle t, \sigma\rangle)$
  - $len(M_1) \leq i_j \wedge i_j \neq len(M_1) \wedge mark([x \mapsto N]index(t - len(M_1), M_2), \sigma)$
    $\Rightarrow mark(M, \langle t, \sigma\rangle)$

The following lemmas are analogous to lemma 14.1.8 to 14.1.10 of Barendregt's book on the lambda calculus [6]. Restricted to pure lambda-terms, the statements are identical to the corresponding lemmas for the labelled lambda calculus. The proofs are quite similar to the corresponding proofs for the labelled lambda calculus and are omitted, with $T(M)$ playing the role of $d(M)$ in the proofs in the case of the labelled lambda calculus.

Lemma 95 *Let $M \in SN$. Let $M^\star \xrightarrow{*}_l R$, where $R \in Terms_2 \wedge index(s, R) = \lambda\langle y_1 \ldots y_k\rangle.P$, for some $s, k \in \omega$. Then,*

- $(\exists R_1 \in Terms_2)\ [M \xrightarrow{*}_l R_1 \wedge R_1^\star \xrightarrow{*}_l R]$
- *Let $f(s) = k$. Then, one of the following hold.*
  - *$index(k, R) = \lambda\langle y_1 \ldots y_k\rangle.P_1\ \wedge P_1^\star \xrightarrow{*}_l P$*
  - *$index(k, R) = T$, where $T \in T_2$. Then, $mark(R, \langle s, \langle 0, 0\rangle\rangle)$.*

Lemma 96 *Let* $M \in BC^{\omega}\bot$. *Let* $mark(M, \ )$ *be defined such that,* $mark(M, \sigma)$ *is true for exactly one* σ*, where* $F_M(\sigma)$ *is defined. Let* $M \xrightarrow{*}_l R$*, where* $R \in Terms_2$. *Let* $index(t, R) = (\lambda\langle y_1 \ldots y_k\rangle.P)^n$. *Furthermore, let* $mark(R, \langle t, \langle 0, 0\rangle\rangle)$ *be true, where* $mark(R, \ )$ *is defined by using induction on the definition 66. Then,* $n \leq label(F_M(\sigma))$.

Lemma 97 $M \in SN \Rightarrow [x \mapsto \bot]M \in SN$

Lemma 98 $M, N \in SN \Rightarrow [x \mapsto N]M \in SN$

Theorem 7 *(Strong normalisation)*

Every reduction starting from a completely labelled term (M, I) *terminates.*