RICE UNIVERSITY

Universal Domains For Sequential Computation

by

Ramarao Kanneganti

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Robert S. Cartwright, Chairman
Professor of Computer Science

Matthias Felleisen
Professor of Computer Science

John C. Polking
Professor of Mathematics

Houston, Texas

May, 1995

Universal Domains For Sequential Computation

Ramarao Kanneganti

Abstract

Classical recursion theory asserts that all conventional programming languages are equally expressive because they can define all partial recursive functions over the natural numbers. However, most real programming languages support some form of *higher-order* data such as potentially infinite streams, lazy trees, and functions. Since these objects do not have finite canonical representations, computations over these objects cannot be accurately modeled as ordinary computations over the natural numbers.

In my thesis, I develop a theory of higher order computability based on a new formulation of domain theory. This new formulation interprets elements of any data domain as lazy trees. Like classical domain theory, it provides a *universal domain* **T** and a *universal language* KL. A rich class of domains called *observably sequential domains* can be specified in **T** with functions definable in KL. Such an *embedding* of a data domain enables the operations on the domain to be defined in the universal language. Unlike embeddings in classical domain theory, embeddings in **T** retain enough computational information to separate terminating and non-terminating computations. An important practical consequence of this embedding is the fact that the definitions of program operations are effective, implying that denotational language definitions expressed in this framework are effective interpreters.

Acknowledgments

I am deeply thankful to:

– Robert "Corky" Cartwright, for making this thesis into a reality.

– Matthias Felleisen, for making graduate school fun and lively.

– Ivy Jorgensen for helping me deal with the rules and regulations.

– Dorai and Kishore for all the helpful discussions.

– NSF for the generous support on the grant CCR-9122518.

– Madhavi for her patience and understanding, her companionship.

Contents

Illustrations

Chapter 1

Introduction

1.1 Motivation

Classical recursion theory [Chu36, Kle36, Tur37] asserts that all conventional programming languages are equally expressive because they can define all partial recursive functions over the natural numbers. This statement, however, is misleading because real programming languages enforce a more abstract view of data than bitstrings. In particular, most real programming languages support some form of *higher-order* data such as potentially infinite (input and output) streams, lazy trees, and functions. In contrast to conventional data objects like numbers, characters, arrays, and lists, *higher-order* data objects do not have finite canonical representations. As a result, computations involving higher-order objects cannot be modeled as ordinary computations over the natural numbers. The integrity of higher-order data abstractions critically depends on the fact that programs obey constraints on how higher-order representations are manipulated. Since classic recursion theory ignores these constraints, it does not address the expressiveness of languages that manipulate higher-order data.

To assess the expressiveness of real languages, we need to develop a theory of computability that includes higher-order data. To be comprehensive, this theory must accommodate all of the data objects that occur in conventional languages, including ordinary finite data objects and higher-order data objects such as lazy trees and functions. To be useful, the theory must provide the same abstract view of data that the language provides.

In the literature on program specification, several frameworks have been developed for specifying data domains containing finite objects as term algebras [JG76, Wan88], but none of them addresses the issue of infinite (higher-order) objects satisfactorily. Infinite data objects, such as functions, play an essential role in advanced programming languages. For example, functions are indispensable in defining abstractions that are not built into the language. Some common examples include input/output streams and lazy lists. Functions are also required to assign a natural meaning to program text as data.

The literature on program semantics has developed a framework for specifying higher order data called *Domain Theory*. Domain theory provides a *universal* domain U, a partially ordered set of data objects where the ordering relation is based on the information content of the data objects. Infinite objects have infinitely many smaller objects below them in the ordering. Within the universal domain, every data object is identified by its finite approximations, and computation on an element is performed by examining its finite approximations and producing the finite approximations of the result. In conjunction with the universal domain, domain theory provides a universal language *LAMBDA* for expressing operations on the domain. Any "reasonable"[1] data domain can be defined as an embedded subset of the universal domain using a function in *LAMBDA*. All the computable functions on the original domain can be described on this embedding using *LAMBDA*.

Even though Domain Theory is a general framework for defining program data domains, it is most commonly used to describe the semantics of languages. In this context, both the programming language syntax and the domain of program denotations are embedded in U. A denotational definition of a programming language is a functional program written in *LAMBDA* that maps source program phrases into their meanings (denotations), which are data objects in the universal domain. The universal domain U has the key property that the domain $U \rightarrow_c U$ of continuous func-

[1] Where "reasonable" depends on the specific choice of U.

tions over U is isomorphic to an embedded subset (subspace) of U. Consequently, the continuous functions over U can be interpreted as elements of U. Every computable function[2] in $U \rightarrow_c U$ can be defined in LAMBDA. Thus, using Domain Theory we can specify higher-order data along with the operations on that data.

1.2 Problem Statement

At first glance, it would appear that Domain Theory has solved the problem of data domain specification and higher-order computation. However, the classical theory suffers from a serious problem: the operations specified using *LAMBDA* are not effective. The model of computation underlying *LAMBDA* is too abstract to simulate the behavior of real programs faithfully. In particular, *LAMBDA* does not distinguish between terminating and non-terminating computations: the evaluation of an expression M in *LAMBDA* simply enumerates all finite approximations to the data object denoted by M. Classical universal domains do not have any finite maximal elements. As an implication, a programmer working in the universal framework expecting an answer such as 42 would only see an answer still being computed after the computation produced the representation for 42. There is no way to see that the computation does not produce further results as the representation cannot indicate that information. Hence, even if the computation produces an answer, it never terminates.

The failure of classical domain theory to preserve maximality in domain embeddings prevents denotational definitions from serving as abstract interpreters. As a result, practical language definitions must include a supplementary operational definition, which may not be consistent with the denotational definition. Ideally, we would like to define the semantics, including termination behavior, for any programming language P by defining a meaning function for P in the universal language.

[2]A function is computable iff it has a recursively enumerable graph.

In the absence of proper treatment of program termination, the universal language cannot serve as a *universal, higher-order* programming language. If the universal domain does not capture the termination behavior of computations, then denotational definitions can only be used as partial specifications of the semantics of languages. They must be supplemented by more explicit operational definitions.

The goal of the research presented in this thesis is to reformulate domain theory so that termination of computations is preserved when they are embedded in the universal domain.

1.3 Contributions

This thesis contributes to both our theoretical and practical understanding of computation. At a theoretical level, it provides a comprehensive framework for expressing higher order sequential computations and identifies a universal language for this form of computation. At a practical level, it provides a framework for reasoning about constructs in programming languages, and identifying the constructs necessary to make the languages computationally *complete*, *i.e.*, capable of expressing any computable operations on the underlying domain.

This thesis contributes to the theory of computation by developing a new formulation of domain theory that is universal for a rich class of domains called *observably sequential domains*. Denotational definitions expressed in this framework are effective; they can be executed by an interpreter written for the universal language. In the classical domain theory, denotational definitions do not have this property.

To define an arbitrary computation over a data domain, it must be defined in the theory. Our formulation of domain theory comes with a language to define the data domains within the universal domain. Thus, this thesis provides a data definition language for higher-order elements.

At a practical level, the thesis provides new models of computation for programming languages. These models do not contain the extraneous "parallel" functions such as *parallel or*[3] that are excluded from *all* practical programming languages. Thus, programming languages using this framework for underlying model of computation, can formulate sequential constructs to make the language complete with respect to the new models. We present a study of one such language SPCF [CF91, CF92][4] in this thesis. We also show how our reformulation of domain theory facilitates the invention of new constructs to make the languages more expressive.

Aside from these fundamental contributions, we develop new techniques to describe domains. We also develop techniques to show full abstraction results for various languages. In particular, we study the language SPCF within our framework and provide a proof of computational completeness. This proof technique is based on the computational completeness of the universal language.

1.4 Overview of the thesis

The thesis is organized as follows. In Chapter 2, we discuss the previous formulations of domain theory and trace their inadequacy to continuous function space. We propose a solution based on "sequential computation".

In Chapter 3, we refine the concept of sequential computation by defining a new class of domains, called OS-domains. These domains are used to describe a new class of functions, called OS-functions. These OS-functions provide a model for sequential computation. The important technical result of this chapter is that the category of OS-domains are closed under OS-function construction.

In Chapter 4, we use OS-domains to provide a model for PCF, a lambda-calculus based language. To this end, we use the standard method of forming a cartesian closed category of OS-domains with OS-functions. The main result of this Chapter is the formation of such a ccc.

[3] A boolean function that needs to evaluate both the arguments in parallel to produce the answer. The full description is given in the next chapter.

[4] SPCF stands for sequential PCF [CF92]. PCF is a language based on simply typed lambda calculus [Plo77].

Chapter 5 describes universal domains based on OS-domains. These universal domains satisfy the goal of the thesis, namely, providing a satisfactory model of computation where termination properties are not ignored. We show that any given OS-domain can be embedded in a universal OS-domain such that the desired properties of computation are preserved. We also show that this solution does not suffer from the technical problems of an earlier attempt to formulate a universal higher-order programming language [CD88].

In Chapter 6, we provide a language KL for the universal domain $\mathbf{T}$. We give both the operational and denotational semantics for this language and show that the denotational definition is adequate and fully abstract. In this language the terminating computations have finite denotations. Finally we show that the language is universal, *i.e.*, it can define any computable element or function of the universal domain.

We provide a case study of these domains in Chapter 7. In this chapter, we study a language called SPCF, a sequential extension of PCF. Using the Cartesian Closed Category (ccc) of OS-domains and OS-functions, we provide an adequate model for SPCF. In addition, we use the techniques from the universal domain framework to show that this model is also fully abstract. The main technical result of this section is a novel approach to prove full abstraction using computation completeness.

In Chapter 8, we briefly summarize the contributions of the thesis. We discuss two different ways to extend this thesis–designing a data definition language and designing an extension of Pure Scheme.

Chapter 2

Background

In this chapter, we review classical domain theory and its important definitions. We discuss the failings of the classical theory from the perspective of computability, trace them to continuous function space construction, and propose a new domain theory based on a different function space construction.

2.1 Models of Higher Order Computation

In the literature on the theory of computability, there are several models of computation. Classical recursion theory [Tur37, Chu36, Kle36] asserts that all models are equivalent. These models of computation are based on encoding finite inputs into a string over a given alphabet. For example, the Turing Machine encodes every input into a finite string and generates a finite string as the answer. This approach works for many common kinds of inputs including natural numbers, character strings, arrays of numbers, and finite trees.

However, most programming languages include non-finite data. The most familiar examples are functions, I/O, and streams. Every language supports stream-oriented input and output with operations that incrementally read and write data items. Using these operations, non-terminating programs can read and write infinite streams of data. Even though most programming languages do not directly support higher-order[5] data, they may have constructs to simulate such data. For example, most languages support the concept of file I/O, which can be thought of as an infinite

[5]By higher-order data, we mean the data that may have an infinite representation, e.g.: Real numbers, Functions.

stream. Thus, providing a satisfactory model for computing over infinite data is essential to understanding the behavior of *practical* programs.

Unfortunately, the model provided by the classical computational theory cannot easily be extended to deal with infinite data. As an example, consider functions over natural numbers as inputs to a program. Since functions, in general, cannot be represented by finite strings, it would appear that Turing Machines cannot compute over these functions. However, we can encode functions as indices of Turing Machines that compute them. Thus, every computable function corresponds to a Turing Machine index. As a result, a computation over an infinite computable function is reduced to computation on a Turing Machine index.

Nevertheless, this approach suffers from a serious problem: there is no unique index for a function. In fact, there are always infinitely many Turing Machines that compute the same given function. Moreover, it is undecidable if two indices represent the same function. This situation is analogous to determining if two programs compute the same function. Such a representation clearly violates the principle of data abstraction since the different representations for the same function are distinguishable.

Any programming language convention that we devise to solve this problem reduces to imposing a discipline on the operations for manipulating the representations of functions. That is, programs can be restricted to using a set of abstract operations over representations. Such a *type discipline* hides intensionality, but does nothing to provide a semantics that is extensional. Moreover, such attempts produce complex systems where reasoning becomes difficult. In addition, these systems do not scale well; they cannot be adapted to more complex data such as functions over functions.

Apart from exposing the intensionality of the representation, this theory cannot handle incrementality of I/O. When the input is infinite, we cannot assign it a Turing Machine index before the program, as the input is not yet known. Similarly, if the program produces incremental output, it would not be possible to describe it using a Turing Machine index until the program terminates.

A solution to these problems is to propose a different model of computation. To make this model work, we should make sure that it has at least the following properties:

1. The model should include all the computable functions of the traditional recursion theory.

2. The model should be able to describe infinite elements without loss of abstraction or incrementality.

3. There should be a natural relation between the computation over higher order elements and the traditional computation over natural numbers.

Domain theory provides such a framework.

2.2 Domain Theory

Models from classical recursion theory focus on natural numbers and other forms of data that have canonical representations as numbers such as strings, arrays, and finite trees. That is, the domain and the range of the functions in the classical recursion theory are denumerable sets such as $\mathbf{N}$ or Σ^*. In contrast, *Domain Theory* focuses on more complex forms of data organized in algebraic structures called *domains*. These domains are "completions" of partially ordered sets of approximations that describe potentially infinite data objects. These approximations can be as simple as natural numbers or as complex as finite functions over functions on natural numbers. Classical domain theory provides a framework to construct these domains.

In addition to describing the domains, this framework is also useful for defining semantics of languages that operate over these domains. For this purpose, domain theory includes a universal language that can express computations over the domains specified. In the following sections, we will expand further on this topic.

2.2.1 Data Domains

Domains are structures consisting of a set of *elements* and an *order relation* over this set. Informally speaking, each element of a domain contains some information. For example, the number 3 in the domain of natural numbers "contains" the information that it is equal to the value 3. Thus the element 3 completely identifies itself. The order relation over the elements is defined by the information content of the elements. This ordering of the elements has nothing to do with the numerical ordering.

Let f be a function that operates over the natural numbers. Let the input to f be a black box that yields "information" about the input number. The black box could be thought of as a function that generates more and more information about the input. If a black box b_1 generates an element 1, it has provided complete information regarding the input. That is, it cannot generate any other number, as it can represent only one number. In this case, the black box b_1 represents 1. Thus, the function f gathers information about the input from this black box and generates more and more information about the output. Therefore, the result of the computation is a black box representing a number, hence an element of the domain.

Since there is a black box that may yield no information (corresponding to the function that never terminates with the answer), the domain should contain an element with zero information content. This element is traditionally referred to as $\perp$ (pronounced "bottom").

Example 2.1 *Natural Number Domain* $\mathbf{N}_\perp$ is the domain described in the preceding paragraph. The domain consist of the set of natural numbers and an additional element $\perp$. The information ordering is represented in Fig. 2.1 as a *Hasse diagram*. Any domain with such information ordering, *i.e.*, where $x \sqsubseteq y$ implies either $x = y$ or $x = \perp$ is called a *ground domain* or a *flat domain*. ■

Example 2.2 *The Boolean Domain* $B_\perp$ is described in Fig. 2.2. It has two non-bottom elements: tt and ff. We will use this domain in subsequent chapters.

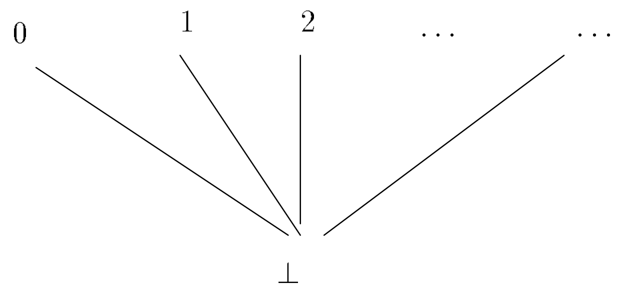

Figure 2.1 Natural Number Domain: $\mathbf{N}_\perp$

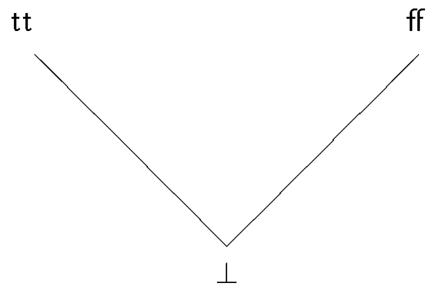

Figure 2.2 Boolean Domain

■

Example 2.3 The *Product Domain* $B_\perp \times B_\perp$ consists of all pairs of elements (b_1, b_2) where $b_1, b_2 \in B$. The domain is ordered pointwise: $(a_1, a_2) \sqsubseteq (b_1, b_2)$ iff $a_1 \sqsubseteq a_2$ and $b_1 \sqsubseteq b_2$. This domain is pictured in Fig. 2.3. ■

The preceding examples provide the basic domains. In the later parts of this chapter, we will show how to construct more complex domains out of simpler ones. Before describing those constructions, we need to formally define domains.

Domains can be formally defined in two different ways: axiomatic and constructive. The axiomatic method identifies when a given partial order is a domain. The constructive method specifies how to build the domains from partial orders with some additional properties. If the procedure is followed, the resulting structures turn out to be domains in axiomatic sense. In the following section, we provide an axiomatic

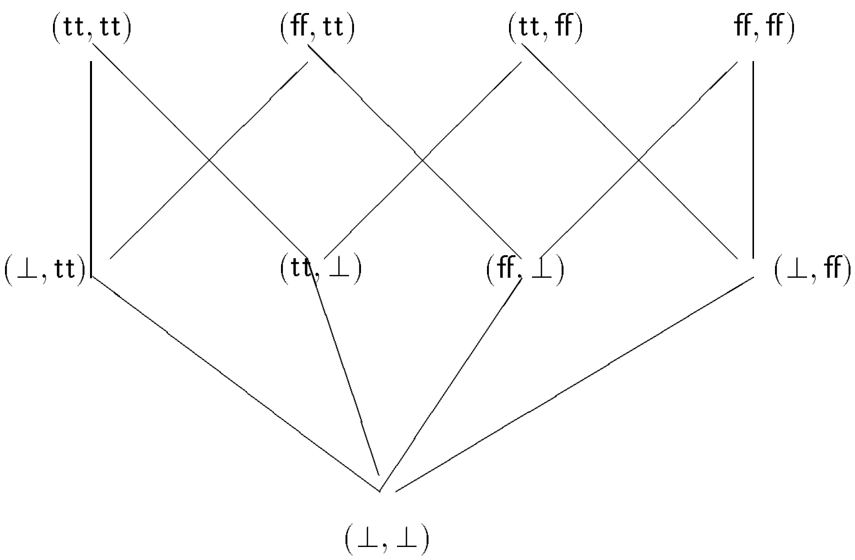

Figure 2.3 Product Domain $B_\perp \times B_\perp$

definition for domains. In section 3.2, we give the complementary constructive specification of domains.

2.2.2 Axiomatic Definition of a Domain

These axiomatic definition of domains can be found in standard literature on domain theory [Sco76, Plo78].

Definition 2.1. (*Partial Order*) A *partial order* $\mathcal{S}$ is a pair $\langle S, \sqsubseteq \rangle$ consisting of a set S of objects, called the *universe*, and a binary relation $\sqsubseteq$ over S such that $\sqsubseteq$ is *reflexive*, *antisymmetric*, and *transitive*. If $a \sqsubseteq b$, we say that a *approximates* b. An element a *precedes* b, written as $a \prec b$, if $a \sqsubset b$ (that is, $a \neq b$ and $a \sqsubseteq b$) and $a \sqsubseteq c \sqsubseteq b$ implies $a = c$ or $b = c$. ∎

An example of a partial order is the order $\leq$ on natural numbers. That is, $\langle \mathbf{N}, \leq \rangle$ forms a partial order. In that partial order, $1 \prec 2$, while $1 \not\prec 3$ because 1 is not immediately below 3.

Example 2.4 The pair $\langle N, \sqsubseteq_m \rangle$ is a partial order where $N = \mathbf{N} - \{0\}$, and

$$n \sqsubseteq_m m \iff m \; mod \; n = 0, , i.e., \text{ n divides m.}$$

It is easy to verify that this pair is a partial order. ∎

Definition 2.2. (*Bounded Set*) Let $\mathcal{S} = \langle \mathcal{S}, \sqsubseteq \rangle$ be a partial order. Then, a set $R \subseteq S$ is called *bounded* if $\exists u \in S : \forall r \in R, r \sqsubseteq u$. ∎

Definition 2.3. (*Least Upper Bound*) If a set A is bounded, a least upper bound of A is defined as the least element r bounding A, *i.e.*, $\forall a \in A : a \leq r$, such that if there is an $s \in A$ that bounds A then $r \leq s$. ∎

The preceding definition defines a least upper bound of a set. We can show that such a least upper bound, if it exists, is unique.

Proposition 2.4 Let $(S, \sqsubseteq)$ be a partial order and A be a subset of S. Then, if A has a least upper bound, it is unique.

Proof. Let a_1 and a_2 be two least upper bounds of the set A. By the definition of least upper bound, $a_1 \sqsubseteq a_2$ and $a_2 \sqsubseteq a_1$. Therefore, $a_1 = a_2$. □

For example, in the partial order $\langle N, \sqsubseteq_m \rangle$, the set $\{4, 6\}$ is bounded by 24. The least common multiple of 4 and 6, which is 12, is the least upper bound of this set.

It is clear that the least upper bound (*lub*) exists only for bounded sets. However, not all bounded sets have lubs.

Notation: We use $\perp$ to denote the least upper bound of the empty set. If the set $\{a, b\}$ is bounded we use the notation $a \uparrow b$. If $\{a, b\}$ is not bounded, we say that a and b are *inconsistent* and denote this relation by $a \# b$. We denote the least upper bound of a and b, if exists, as $a \sqcup b$. The least upper bound of a set A is designated as $\bigsqcup A$.

Definition 2.5. (*Chain, Directed Set*) A subset R of a partial order is a *chain* iff it is totally ordered:

$$\forall r_1, r_2 \in R : r_1 \sqsubseteq r_2 \vee r_2 \sqsubseteq r_1.$$

A subset R of a partial order is *directed* iff every finite subset of R has an upper bound in R. ∎

It is clear that any chain is a directed set. For any countable directed set R, we can find a chain $R' \subseteq R$ such that $r \in R$ implies $r \sqsubseteq r' \in R'$ as follows. Since R is countable, we can enumerate the elements of R as $r_0, r_1, \ldots$. We can filter out some elements out of that enumeration by using the criterion that we retain an element r_n only if for all $j < n$, $r_j \sqsubseteq r_n$. Such a scheme automatically yields a chain. It is easy to verify that the lub of the chain R' is equal to that of the set R.

Example 2.5 In the example 2.4, the set $\{4, 6\}$ is not directed. However, $\{4, 6, 12\}$ is directed, but not a chain. The set $\{4, 8, 16\}$ is a chain. ∎

Definition 2.6. (*Complete Partial Order(cpo)*) A partial order $\langle S, \sqsubseteq \rangle$ is said to be complete iff

- there is a least element $\perp \in S$, that is, for any element $s \in S$, $\perp \sqsubseteq s$,
- every directed set $R \subseteq S$ has a least upper bound r in S.

∎

Notation: Whenever it is clear from the context, we drop the notation $\langle S, \sqsubseteq \rangle$ and use just S. If there are more than two partial orders, we suffix $\sqsubseteq$ or $\leq$ with appropriate subscripts.

It is obvious that the examples we presented previously are cpo's. An example of a partial order that is not a cpo is the set of natural numbers with the partial order as $=$. This set lacks a least element, hence is not a cpo. This set is not a cpo under the ordering $<$ either; the order is not a reflexive and hence not a partial order. For a different kind of example of a non-cpo consider the following: let S be the set of natural numbers ordered under $\leq$. Clearly 0 is the $\perp$ element for this set. The set S is a directed set, yet does not have a lub in S. Hence it is not a cpo.

An arbitrary cpo does not have enough structure to perform incremental computation; we need to add the computational structure of *finite* elements to the cpos to make them domains. These elements provide a unique way of identifying a domain.

Definition 2.7. (*Finite element*) An element b of a cpo $\langle D, \sqsubseteq \rangle$ is finite iff for every directed set R such that $b \sqsubseteq \bigsqcup R$, there is an element $r \in R$ such that $b \sqsubseteq r$. ∎

Example 2.6 All the elements of $\mathbf{N}_\perp$ are finite elements trivially. Consider a more complex domain, $(\mathbf{N} \cup \{\infty\}, \leq)$. In this domain $\bigsqcup \mathbf{N} = \infty$, but $\nexists n \in \mathbf{N} : \infty \sqsubseteq n$. Therefore, ∞ is not a finite element. ∎

Before defining what a domain is, we will define the property of ω-algebraicity. Intuitively, this property implies that every element can be described by finite elements.

Definition 2.8. (ω-*Algebraicity*) A cpo $\langle D, \sqsubseteq \rangle$ is algebraic if every element is the least upper bound of its finite approximations. It is ω-algebraic if the set of finite approximations is countable. ∎

Now, we are ready to define a domain.

Definition 2.9. (*Domain*) A cpo $\langle D, \sqsubseteq \rangle$ is a *domain* if it is ω-algebraic, and every bounded set has a lub in the cpo. ∎

It is apparent from the preceding definition that finite elements play a special role in a domain. Any element of the domain can be defined as the set of its finite approximations. Therefore, it is possible to identify a domain uniquely with its finite elements. Infinite elements appear as least upper bounds of their finite approximations. We make use of this fact when we describe the computation over a domain.

It may appear that this definition of a domain is unnecessarily general. The only kind of domain we used in classical recursion theory is the domain $\mathbf{N}_\perp$. It is natural to ask how more complex domains come up in practice. The answer is that we inductively define these complex domains. These domains model different kinds of data in programming practice. Again, these constructions can be found in the standard literature on domain theory.

2.2.3 Domain Constructions

Given a collection of basic domains, we can construct more complex domains domains using a variety of domain construction methods.

Definition 2.10. (*Lifting*) For a domain $(D, \sqsubseteq)$, define the *lifted* domain $D_{\odot}$ as a domain $(D', \sqsubseteq')$ where

- $D' = D \cup \{\perp'\}$,
- $\sqsubseteq' = \sqsubseteq \cup \{(\perp', x) \mid x \in D'\}$.

∎

It is easy to verify that $D_{\odot}$ is a domain. The fact that newly formed domain is a partial order can be verified as follows. The element $\perp'$ approximates every element of the set D'. In addition, the element $\perp'$ is the only new finite element. Therefore, all the finite elements of $D_{\odot}$ are countable. To show that an element of D' is the lub of its finite approximations, consider an element $a \in D'$. If $a = \perp'$, we are done. If not, a must belong to D also. Therefore $a = \sqcup A$, where A is the set of finite approximations of a in D. Since $\perp'$ approximates every element in D', we can extend the equation to $a = \sqcup(A \cup \{\perp'\})$, which is the required condition for $D_{\odot}$ to be a domain.

Definition 2.11. (*Sum*) Given two domains $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, we define the (smashed) *sum* domain $A \oplus B$ as follows:

- The set of elements is $(1 \times (A - \{\perp_A\})) \cup (2 \times (B - \{\perp_B\})) \cup \perp_{A \oplus B}$. That is, we remove the $\perp$ elements from both A and B and add an additional $\perp$ element.
- The ordering relation is defined as follows: $\perp_{A\oplus B} \sqsubseteq x \in A \oplus B$, $(1, x) \leq (1, y)$ iff $x \sqsubseteq_A y$, and $(2, x) \leq (2, y)$ iff $x \sqsubseteq_B y$.

∎

To show that the preceding construction yields a domain, we need to show that it is a complete partial order and it is ω-algebraic. It can easily be verified that the new ordering is a complete partial order. The finite elements of the new domain are $\{(1,a),(2,b)\}$ where a is a finite element in A and b is a finite element in B. Therefore this set is countable. For algebraicity, let $(1,a)$ be an element of the new domain. Since a is an element of the domain A,$a = \sqcup A$, where A is the set of finite approximations to A. Therefore $(1,a) = \sqcup\{(1,a_i)\}$ where a_i belong to A. Since a_i is a finite element in A, $(1,a_i)$ is a finite element in $A \oplus B$. Therefore $(1,a)$ is the least upper bound of its finite approximations. Similarly we can prove for an element $(2,b)$ also. Hence, $A \oplus B$ is a domain.

Definition 2.12. (*Product*) Given two domains $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, we define the *product* B[6] domain $A \times B$ as follows:

- The set of elements is $\{(a \times b) \mid a \in A, b \in B\}$.
- The ordering relation is $(a_1, b_1) \sqsubseteq_\times (a_2, b_2)$ iff $a_1 \sqsubseteq_A a_2$ and $b_1 \sqsubseteq_B b_2$.

∎

To show that the $\times$ construction yields a domain, we first observe that the underlying order is a partial order. It also can be verified that the element $(\bot_A, \bot_B)$ is the least element of the partial order. Further, if a set $\{(a_i, b_i)\}$ is bounded, then the set $\{a_i\}$ is bounded in A and the set $\{b_i\}$ is bounded in B as well. Therefore, the set $\{(a_i, b_i)\}$ has a least upper bound (a, b) where a is the least upper bound of $\{a_i\}$ and b is the least upper bound of $\{b_i\}$. Hence the preceding definition describes a complete partial order.

To show the ω-algebraicity, consider the set of finite elements of the domain $A \times B$. This set is the cartesian product of all the finite elements of A and B. Thus, the set of finite elements in the product domain is countable. To show the algebraicity,

[6]It is not the only product domain we can build. For a full description, refer to [CD82].

consider an element (a, b) in the product domain. Since $a = \bigsqcup\{a_i \,|\, a_i \sqsubseteq_A a\}$ and $b = \bigsqcup\{b_i \,|\, b_i \sqsubseteq_B b\}$, the element $(a, b) = (\bigsqcup\{a_i\}, \bigsqcup\{b_i\}) = \bigsqcup\{(a_i, b_i)\}$. Therefore, the product domain is algebraic.

The preceding definitions provide methods to build domains from simple domains such as $\mathbf{N}_\perp$. It can easily be verified that each of the above methods results in a domain. Therefore, constructions such as $\mathbf{N}_{\perp_\odot} \oplus (\mathbf{N}_\perp \times \mathbf{N}_\perp)$ are domains. For a semantic description of different domain constructions, refer to [CD82]. For a textbook description in a categorical setting, see [Gun92].

It is easy to observe that starting from a simple domain such as $\mathbf{N}_\perp$, one can never construct a domain with infinite elements using these constructions. However, there is an important construction that generates infinite elements starting from a domain with only finite elements. This construction generates the function space over two domains.

Function Domain Construction

In domain theory, computation over a domain proceeds incrementally. In a function application, the function incrementally gathers information about its input and adds information to its output. This view is further expounded in the section 3.1.

In this computational model, a function can only generate more information about the output as the computation proceeds; it cannot take back any output that it has already generated. Hence, the output information keeps on increasing (or remains the same) as more and more information about the input is gathered. This property is called *monotonicity*.

Definition 2.13. (*Monotonicity*) A function $f \in A \to B$ is monotonic if $x \sqsubseteq_A y \Rightarrow f(x) \sqsubseteq_B f(y)$. ∎

In this incremental computation, a function can never see all the information regarding an infinite element. It gathers more and more information about the input and generates more and more information about the output. If the information to be gathered is infinite, the process never terminates. However, we can expect the behavior of the function on an infinite input as the limit point of its behavior on all the finite approximations. This notion, called *continuity*, is a simple extension of monotonicity.

Definition 2.14. (*Continuity*) A function $f \in A \to B$ is *continuous* if a set X is directed, then $f(\sqcup X) = \bigsqcup\{f(x) \mid x \in X\}$. ∎

This is the central definition of domain theory. The notion of computation in domain theory is rooted in the concept of continuity. There are several implications of this concept.

- A continuous function is monotonic. It can easily be seen that if $a \sqsubseteq b$ then $b = \bigsqcup\{a, b\}$, hence $f(b) = f(a) \sqcup f(b)$. Therefore, $f(a) \sqsubseteq f(b)$.

- If a_1 and a_2 are consistent, so are $f(a_1)$ and $f(a_2)$. That is, the function is constrained to produce consistent information for the inputs a_1 and a_2.

- For an infinite element, the behavior of the function is completely described by the behavior of the function on its finite approximations. That is, if a is an infinite element, then there is a set of finite approximations $X \subseteq A$ such that $\sqcup X = a$. Obviously X does not contain a. And, $f(a)$ is determined completely by the set $\{f(x) : x \in X\}$. Thus, continuity gives a handle on infinite elements.

Any function over $\mathbf{N}$ can easily be extended to form a continuous function over $\mathbf{N}_\perp$, by adding $(\perp, \perp)$ to its graph. Hence, the extension of an uncomputable function is a continuous function on $\mathbf{N}_\perp$. Continuity does not imply computability! Later we will show how the computable subset of continuous functions can be defined using traditional recursion theory.

An important property of domain theory is that the set of computable functions can be interpreted as data for higher order computation. This property is due to the fact that the set of continuous functions form a domain. The following theorem states the result precisely.

Theorem 2.15 Let $A \rightarrow_c B$ be the set of continuous functions over the domains A and B. Then, that set ordered under $\sqsubseteq_\rightarrow$ where $(f, g) \in \sqsubseteq_\rightarrow$ if $\forall a \in A : f(a) \sqsubseteq_B g(a)$, is a domain.

Proof. We will provide only a brief outline for the proof. For a complete proof, see Plotkin's seminal paper [Plo78]. It is easy to confirm that $\sqsubseteq_\rightarrow$ forms a partial order. In addition, the partial order is complete, since there is a least function with graph $\{(x, \perp) \mid x \in A\}$ and, every directed set S od functions in $A \rightarrow_c B$ has the least upper bound $\{(x, \sqcup f(x)) \mid f \in S, x \in A\}$ is continuous.

For algebraicity, we need to show that the set of finite functions is countable and every function in $A \rightarrow_c B$ is a least upper bound of its finite approximations. Consider the set of functions with the graphs $\{(a, b)\}$, each denoted as $s_{a,b}$, where a and b are finite elements in A and B respectively. These functions are called *step* functions. Any finite function is a least upper bound of a *finite* number of such step functions. Moreover, the converse is true: a function is finite only if it is the least upper bound of a finite number of step functions. Therefore the number of finite functions is countable.

Consider an infinite function f. For a finite input a if $b \sqsubseteq f(a)$ then the step function $s_{a,b}$ approximates f. Therefore, if $f(a) = \sqcup b_i$ then the set of functions s_{a,b_i} approximate f. Since $f(a) = \sqcup s_{a,b_i}(a)$, we have the desired result in case of a finite input.

For an infinite input $a' = \sqcup a_i$, since $f(a') = \sqcup f(a_i)$, we can get the same result of the preceding paragraph. $\square$

By using the continuous function construction, we can construct more complex domains with infinite elements starting from simpler domain containing only finite elements.

Example 2.7 Consider the function space $\mathbf{N}_\perp \rightarrow_c \mathbf{N}_\perp$. The function 1+ that adds 1 to every element is an infinite element in this domain. $\blacksquare$

Since the continuous function space construction also yields a domain, we can construct suitable domains to model computation over function spaces. Since the domain of functions itself is a domain, we can define continuous functions over this domain. Thus, it is possible to build domains such as $A \rightarrow_c (B \rightarrow_c (C \times D))$. In some cases the lift, sum, product and function constructions are sufficient to build a model for a given programming language. But in most cases, we require more general techniques to build domains.

Consider a domain D satisfying the equation: $D = \mathbf{N}_\perp \oplus D \rightarrow_c D$. Such "recursive" domains often appear in programming practice. For example, in higher order programming languages, functions are data objects. Hence the domain of computation must satisfy an equation similar to the equation for D. How can we be sure such equations have solutions? We can show that the domain of domains itself is a domain. In this context, equation like $D = \mathbf{N}_\perp \oplus D \rightarrow_c D$ is simply a recursive definition of an element in that domain. This argument can be formalized using the concept of *Universal Domains*.

2.2.4 Universal Domains

As described previously, a universal domain provides a framework for defining arbitrary domains and computable functions over these domains. In conjunction, with an accompanying universal language, a universal domain provides the machinery required to define the meaning of programming constructs. With the right kind of universal domain, it is even possible to generate an effective interpreter for a given language from the definition written in the universal language.

A universal domain U and the corresponding universal language L_U provide a framework for a certain class of domains. That is, when we call a domain universal, it is universal for a class of domains, *i.e.*, we can *embed* any domain belonging to that class in the universal domain.

To define universal domains formally, we must define what embedding means. Informally, embedding a domain A in B means, describing A in terms of B. If A can be embedded in B, there is an isomorphic image of A within B specified as a constraint on B. Such an isomorphic domain is called *subdomain*.

Definition 2.16. (*Subdomain*) A domain $(A, \sqsubseteq_A)$ is a subdomain of a domain $(B, \sqsubseteq_B)$ iff

- $A \subseteq B$,
- $\sqsubseteq_A = \sqsubseteq_B \cap (A \times A)$.
- $a_1 \# a_2$ in A implies $a_1 \# a_2$ in B.

∎

A domain A is embedded in a domain B by finding an isomorphic image A' that is a subdomain of B. The embedding of domains requires mapping elements of one domain into another. To facilitate the definitions of such mappings, we introduce the concept of *effective presentation* of a domain. An effective presentation names each finite element of a domain as a natural number such that we can gather the same information from these natural numbers as the underlying partial order.

Definition 2.17. (*Presentation, Effective Presentation*) A *presentation* α of a domain D is an enumeration of the finite elements of D that maps $\perp_D$ to 0. It is said to be *effective* iff the following relations are recursive:

- The binary relation Con defined by

$$Con(i, j) \Leftrightarrow \exists k[\alpha_i \sqsubseteq \alpha_k \wedge \alpha_j \sqsubseteq \alpha_k .]$$

- The ternary relation Lub defined by

$$Lub(i, j, k) \Leftrightarrow \alpha_k = \sqcup\{\alpha_i, \alpha_j\} .$$

∎

Definition 2.18. (*Universal Domain*) A domain U is universal for a class of domains S iff every domain D in S can be embedded in U. Moreover, if the domain D is described effectively, then the embedding is computable. That is, corresponding to the recursive relations Con, Lub over D, there are recursive relations Con' and Lub' over the subdomain. ∎

Corresponding to a universal domain, there is a universal language. This language lets us define all the computable operations over the universal domain. The reason we require the language is that we can define the domains as functions over the universal domains. Every subdomain is described as a retraction. A retraction is an idempotent function over the universal domain. That is, D is identified by a retraction f_D over U iff

- If $d \in D$, $f_D(d) = d$.
- For every $u \in U$, $f_D(u) \in D$.
- f is continuous.

It is clear that f coerces the elements of U into D. Thus, every domain embedded in the universal domain can be described as a retraction over the universal domain. The universal language lets us define these retractions.

The notion of computability in the universal domain is related to traditional recursion theory. We use the effective presentation of the universal domain to define a computable element of the domain.

Definition 2.19. (*Computable element*) An element d of a universal domain U is computable if the set of finite approximations to d is enumerable, that is, there is a recursive function f_d over the natural numbers that generates all the indices of finite approximations. ∎

If we can embed the function domain $U \to_c U$ into U, we can define the computable functions using the preceding definition.

T^ω – An Example Of A Universal Domain

Gordon Plotkin developed T^ω as a universal domain for the class of *coherent* domains [Plo78][7]. This domain can be described as ω-ary product of the simple boolean domain given in the picture 2.2.

Thus, the elements of the domain are infinite tuples of the form $\langle a_1, a_2, a_3, \ldots \rangle$, where $a_i \in \{\bot, \mathsf{tt}, \mathsf{ff}\}$. An element $\langle a_1, a_2, \ldots, a_i, \ldots \rangle$ approximates $\langle b_1, b_2, \ldots, b_i, \ldots \rangle$ iff $a_i \sqsubseteq_{\text{bool}} b_i$. It can easily be shown that this partial order is a domain. Another way to denote an element of T^ω is as a pair of sets $\langle u, v \rangle$, the first set being the positions with tt and the second set being the positions with ff. Naturally these two sets are disjoint for any given element of T^ω. We use both these notations interchangeably.

The universal language corresponding to T^ω is called *LAMBDA*; it can define all the computable operations over T^ω. This language is given in Fig. 2.4. In this description, we use $\pi_i(x)$ as the i'th projection of x, *i.e.*, the boolean value at i'th position of x.

The function $Fun(x)$ deserves a longer explanation. Fun interprets the elements of T^ω as continuous functions. The full details of Fun are provided in [Plo78]. We provide the brief description in the following paragraphs.

Define the functions f_k as follows. We use the notation $\langle\langle n, m \rangle\rangle$ to denote the standard bijective pairing: $n + 1/2 * (n + m) * (n + m + 1)$. Also, we encode the finite elements of T^ω as b_i where $i = \Sigma_{j \in u} 2^j + \Sigma_{k \in v} 3^k$ where $\langle u, v \rangle$ is the element of T^ω.

$$
\begin{aligned}
f_{\langle\langle n, 2m \rangle\rangle} &= \{(b_n, \langle \{m\}, \emptyset \rangle)\} \\
f_{\langle\langle n, 2m+1 \rangle\rangle} &= \{(b_n, \langle \emptyset, \{m\} \rangle)\}
\end{aligned}
$$

These functions are represented as their minimal graphs, *i.e.*, their continuous extensions provide the full graph. It can easily be seen that these functions are *one-step*

[7] Earlier formulations of universal domains were for the class of lattices only. Scott developed the universal domain $\mathsf{P}\omega$ [Sco76] that can embed all the lattices, that is domains in which all the elements are consistent with one another.

functions. That is they are the functions with minimal graphs that produce a single boolean element in the output position for a finite element.

Now, we define $seg(k)$ as the element $\langle\{k\}, \{n \leq k \mid f_n \# f_k\}\rangle$. Thus, $seg(k)$ provides some information about f_k. $seg(k)$ is the *characteristic* segment of f_k; it is not the complete embedding of f_k in T^ω as such an embedding is infinite. But this segment is sufficient to identify it with respect to the functions embedded so far.

We define $Fun(x)$ as follows:

$$Fun(x) = \bigsqcup\{f_k \mid seg(k) \sqsubseteq x\}$$

Thus, $Fun(x)$ is the least upper bound of all the finite functions with characteristic segments approximating x. Thus, Fun interprets the elements of T^ω as functions.

Function abstraction has the standard meaning given that functions are encoded as elements of T^ω. It follows all the α, β equivalency relations for the λ-calculus.

It can be shown that all the functions defined by *LAMBDA* are continuous. In addition, if an element of T^ω is computable, *i.e.*, its positive and negative sets can be enumerated by a recursive function, then it is definable in *LAMBDA*. Also, the combinator Y is definable in *LAMBDA* as $\lambda x.(\lambda y.x(y(y)))(\lambda y.x(y(y)))$. It can be shown that $Y(f)$ for any continuous f defines a least fixed point for f, *i.e.*, $f(Y(f)) = Y(f)$ and if $f(x) = x$ then $Y(f) \sqsubseteq x$. For the full details of computability in T^ω, refer to [Plo78].

We can show that any coherent domain is isomorphic to a retract within T^ω. For a domain with effective presentation, this retraction function is computable, and hence definable in *LAMBDA*. Thus *LAMBDA* is a universal language.

Example 2.8 Consider the domain consisting only of $\bot$. It can be defined by the function: $Y(\lambda x.x) = \bot$ which takes every element of T^ω to $\langle\emptyset, \emptyset\rangle$. ■

Example 2.9 Consider the domain $\mathbf{N}_\bot$. Each element n can be defined as

$$\langle\overbrace{\mathsf{ff}, \mathsf{ff}, \ldots, \mathsf{ff}}^{n}, \mathsf{tt}, \bot, \ldots\rangle.$$

Unary Functions:

$$\begin{array}{rcl}
\mathsf{tt}* & = & \{(x,y) \mid \pi_1(y) = \mathsf{tt} \wedge \pi_i(x) = \pi_{i+1}(y)\} \\
\mathsf{ff}* & = & \{(x,y) \mid \pi_1(y) = \mathsf{ff} \wedge \pi_i(x) = \pi_{i+1}(y)\} \\
Tl & = & \{(x,y) \mid \pi_i(y) = \pi_{i+1}(x)\}
\end{array}$$

Conditional:

$$if(x,y,z) = \begin{cases} y & \text{if } \pi_1(x) = \mathsf{tt} \\ z & \text{if } \pi_1(x) = \mathsf{ff} \\ y \sqcap z & \text{otherwise.} \end{cases}$$

Application:

$$Apply(x,y) = Fun(x)(y)$$

Abstraction:

$$\lambda x.\tau = \langle u, v\rangle \text{ where}$$

$$\begin{cases} (n, 2m) \in u & \text{if } \pi_m(\tau[b_n/x]) = \mathsf{tt} \\ (n, 2m+1) \in u & \text{if } \pi_m(\tau[b_n/x]) = \mathsf{ff} \\ (n, 2m) \in v & \text{if } \exists n' : b_{n'} \uparrow b_n \wedge \pi_m(\tau[b_{n'}/x]) = \mathsf{tt} \\ (n, 2m+1) \in v & \text{if } \exists n' : b_{n'} \uparrow b_n \wedge \pi_m(\tau[b_{n'}/x]) = \mathsf{ff} \end{cases}$$

Figure 2.4 *LAMBDA*– Universal language for T^ω

The corresponding retraction function can be defined as the least fixed point of the equation:

$$f(x) = if(x, \mathsf{tt} * \perp, \mathsf{ff} * f(Tl(x)))$$

In this definition we are using $\perp$ as the abbreviation for $Y(\lambda x.x)$. The function f coerces the elements of T^ω into the representation of natural numbers. ∎

2.2.5 The Role of Continuous Functions

Continuity is the topological constraint characterizing the incrementality of the computation. The output of a function is simply the enumeration of finite approximations to the answer.

Continuous functions have many useful properties that make them a natural choice for computation over higher-order domains. We list some of these properties as follows:

- Continuous functions provide a natural method to describe computation over infinite elements. The behavior of a continuous function over an infinite input element is the least upper bound of all its finite approximations.

- Continuous functions over a domain form a domain, thus allowing computations over the function space. The set of continuous functions $D \to_c D$ over a domain D can be formally defined by embedding $D \to_c D$ in the universal domain. Therefore, if we model functions on a domain using the continuous functions, we can perform computations over the function space.

- Continuous functions are ordered using the pointwise ordering on the functions based on their extensional behavior.

  Thus, a model based on continuous functions provides an intuitively appealing order-extensional model.

- Continuous functions over the *ground domains* form a superset of computable functions. Thus, continuous functions can capture the notion of Turing computability over the ground domains and extend it to the domains inductively built from those domains using continuous function construction.

- Continuous functions can easily be described using programming language constructs. In fact, the functions definable in a universal language, as defined in classical domain theory, are precisely the continuous functions with r.e. graphs. Thus, the traditional universal languages can compute all the computable continuous functions.

Despite all these advantages, there is growing evidence that the space of continuous functions is too general to describe computation in practical programming languages.

2.3 Why Continuous Functions Are Too General

The continuous function space construction suffers from three important technical problems that make the framework unsuitable for the basis for a model of computation:

1. The universal domains based on continuous functions do not capture the termination behavior of computations in the embedded domains. Since a computation produces the finite approximations to the answer, the only way to indicate that a computation terminates is to generate an approximation that is maximal, *i.e.* , an element that does not approximate any other element. At this point, the computation is complete because it cannot produce any more approximations.

   Unfortunately, none of the classical formulations of universal domains, namely $\mathsf{P}\omega$ [Sco76], T^{ω} [Plo78], and $\mathcal{U}$ have any finite, maximal elements. Hence termination information is not included in the computation. Cartwright and Demers [CD88] addressed this problem by constructing a universal domain ∇ that has finite maximal elements. Moreover, any domain can be embedded in ∇ so that the maximality of finite elements is preserved. However, their extension of the domain theory was not completely successful. In particular, maximality information is not preserved by the continuous function space construction. The following example explains the reason.

   Example 2.10 Consider the boolean domain with tt, ff above the $\perp$. A function $\mathsf{ff} \to_c \mathsf{ff}$ takes a maximal element to a maximal element, yet is not a maximal function. We need additional information about the domain, namely that tt and ff completely cover $\perp$, to decide that the function $\{(\mathsf{tt},\mathsf{tt})(\mathsf{ff},\mathsf{ff})\}$ is a maximal element. This additional information about the function space cannot be obtained from effective presentations of the component spaces. ■

The failure of existing formulations of domain theory to preserve maximality in domain embeddings prevents denotational definitions from serving as abstract interpreters. As a result, practical language definitions must include a supplementary operational definition, which may not be consistent with the denotational definition. Ideally, we would like to define the semantics, including termination behavior, for any programming language P by defining a meaning function for P in the universal language. In the absence of proper treatment of program termination, the universal language cannot serve as the universal *higher order* programming language. If the universal domains do not capture the termination behavior, they can be used only as partial justifications of more effective definitions such as operational interpreters.

2. Continuous functions introduce another serious complication in the data definition framework. The set of continuous function includes deterministic parallel functions such as *por* (parallel-or)[8]. However, these functions do not have analogs in practical programming languages. The presence of such functions in the domain of continuous functions makes the domain contain more computable functions than that can be expressed in conventional programming languages. The functions that cannot be distinguished in the language are distinguished in the domain because of these additional elements. Thus, models based on continuous functions are not *fully abstract*[9] for most languages.

3. The approximation relation on $A \rightarrow_c B$ is contavariant in A yielding a more complex ordering relation than approximation ordering on either A or B. As a result, any formulation of the universal domain D, that accepts $D \rightarrow_c D$ as an embedded subspace must accommodate a more complex family of domains than D.

[8]The full description of *por* is given in example 3.1

[9]A domain is fully abstract for a language if phrases in the language have different denotations iff they can be distinguished in a program context.

Most domains, except for those formed by continuous function construction, are finitely founded, *i.e.*, any finite element has only finitely many elements below it. However, the continuous function space over non-trivial finitely founded domains has infinite elements approximating finite elements, hence is not finitely founded.

Example 2.11 The domain we most often encounter in programming languages is $\mathbf{N}_\perp$. It is a partial order of natural numbers with $\perp$ as the only element that is strictly below any other element.

However, the continuous function domain $\mathbf{N}_\perp \rightarrow_c \mathbf{N}_\perp$ has the following structure: The constant function $f = \{(\perp, 1)\}$ is approximated by an infinite number of functions $f_i = \{(i, 1)\}$ for $i = 0, 1, \ldots$. Thus, a simple flat domain gives rise to a non-finitely founded domain.

■

Example 2.12 Consider a domain D that has an infinite ascending chain $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$. The domain $D \rightarrow_c D$ has an infinite *descending* chain $f_1 \sqsupseteq f_2 \sqsupseteq \ldots$, where $f_i = \{(a_i, b)\}$ and b is a non-bottom element. ■

Since a universal domain has to accommodate all the basic domains along with the domains constructed by the continuous functions, either the universal domain, or the embedding process becomes too complicated. Since we need to embed domains that are not finitely founded and domains that have infinite descending chains, it is difficult to devise universal programming languages. Thus the model using continuous functions becomes complicated by continuous function spaces themselves.

2.4 The Proposal: Sequential Function Construction

Our analysis of domain theory suggests that although the space of continuous functions can be used to provide elegant mathematical models of programming languages, the construction is too general to serve as the basis for a practical formulation of domain theory. Fortunately, there are more restricted function constructions [KP93, Ber79, Cur93, CCF94, CF92] that accommodate all conventional programming languages. To date, they have not been used as the basis for a universal language and domain.

In this thesis, we present a new formulation of domain theory that captures the terminating behavior of arbitrary sequential computations involving higher order data. The new theory is based on the class of functions called OS-functions introduced in [CF92]. This class of functions has been used to provide a fully abstract model for a sequential extension of the simply typed lambda calculus. This thesis generalizes these functions to a broader class of domains.

The core of the new theory is the construction of a universal domain $\mathbf{T}$ and a language KL (short for Kleene's Language) such that:

1. KL can express any computable element or computable observably sequential function of $\mathbf{T}$.

2. Every function expressed in KL is observably sequential.

3. Every OS-function on $\mathbf{T}$ has a fixed point, so that we can define infinite elements using ascending chains of finite approximations.

4. Any domain D satisfying certain topological constraints (identified later) can be embedded by a one-to-one mapping $\delta : D \to \mathbf{T}$ such that:

    - The approximation ordering $\sqsubseteq$, the consistency relation, and the least upper bound relation are preserved.

- The finite maximal elements of the domain D are mapped to finite maximal elements of $\mathbf{T}$.

- There is an OS-function f_D describing the embedding, i.e., $f_D(\mathbf{T})$ is the embedded domain and f_D is idempotent. If the presentation of the domain D is effective, the function f_D is computable, and hence expressible in the language KL.

The culmination of the new theory is the construction of a universal domain $\mathbf{T}^*$ and the corresponding language **KL*** with all the properties listed above for $\mathbf{T}$ and KL plus

- all maximal elements are mapped to maximal elements,

- all finite elements are mapped to finite elements.

The cost of these extra properties is a more complex domain $\mathbf{T}^*$ and a more complex universal domain **KL***.

2.4.1 Implications

The implications of the thesis is as follows:

1. KL and **KL*** are universal programming languages for sequential computation. The precise notion of sequentiality is explored in the thesis. Informally, all constructs that do not require independent threads of computation that *need* to communicate are sequential. By this definition, the cores of all practical programming languages are sequential.

2. KL and **KL*** can be used to define an effective interpreter for a given language $\mathcal{L}$. Unlike previous formulations of domain theory, the semantic specification of $\mathcal{L}$ constitutes an effective interpreter.

   The procedure to derive the interpreter for the sequential language $\mathcal{L}$ as follows:[10]

[10]We assume that the language $\mathcal{L}$ is presented in abstract syntax.

Step 1: Define the syntactic and semantic domains for $\mathcal{L}$ as maximality preserving retracts of $\mathbf{T}$ by writing the corresponding retraction in KL.

This step guarantees that these domains can be embedded to preserve maximality.

Step 2: For each constructor C in the language $\mathcal{L}$, write a function $\mathbf{C}$ in KL that interprets C.

The mapping functions completely specify the semantics of the language $\mathcal{L}$ by mapping the syntactic terms into semantic algebra over $\mathbf{T}$.

Step 3: Assign an operational semantics to the language $\mathcal{L}$ by executing the interpretation $\mathbf{C}$ in KL for each syntactic constructor C. Since the finite maximal elements of the domains are mapped to finite maximal elements of $\mathbf{T}$, any terminating computation in $\mathcal{L}$ terminates in KL as well. Since the operational semantics of KL is fully abstract with respect to $\mathbf{T}$, it follows that the denotational semantics of the language $\mathcal{L}$ agrees with the operational semantics.

The procedure outlined above gives us a method to derive an operational interpreter that is provably in agreement with denotational semantics. In contrast to classical domain theory, the interpretation given by the operational interpreter preserves the termination behavior of computations correctly.

Thus the new domain theory provides a framework where the denotational definition of language $\mathcal{L}$ defines an operational interpreter in the universal language. This operational interpreter is *effective* because it faithfully mimics the computations in the language $\mathcal{L}$. This behavior is possible for two reasons:

- In our domains, answers are denoted by finite, maximal elements. Thus, the interpreter stops computing when it arrives at answer. In the previous universal domains, because of the absence of finite maximal elements, the computations never terminate. Thus, an external examiner must determine that the computation has reached an answer before stopping the computation.

- In our domains, the mappings preserve finiteness and maximality of elements. Thus, an answer in the original domain is mapped to an answer in the universal domain. Therefore, the operational interpreter, computing the translated phrase in the universal domain, stops with the answer.

Thus, the operational interpreter is in full agreement with the denotational semantics of the given language.

Chapter 3

A New Class of Domains

In this chapter, we develop the concept of *observably* sequential computation as an extensional refinement of sequential computation [BC85, KP93]. Since observably sequential computation requires additional structure on the domains, we introduce the class of domains, called OS-domains, in the following section.

3.1 Higher Order Computation

In this section, we briefly review the intuitive model of computation corresponding to continuous functions. By making suitable modifications to that model of computation, we will develop a sequential model of computation.

3.1.1 Continuous Functions

A computation over a (Scott) domain generates progressively better approximations to the final answer. In order to produce the next approximation, the computation, or the function, gathers some new information about the input. We can model the input as an oracle: the input receives a query from the function and answers true or false. Such an answer reveals some new information about the input.

Since each element of the domain can be thought of as an ideal, the set of all its approximations, each query can reveal a new approximation to the input element. Then, the query is of following form:

Does the finite element b approximate the input I?

Such a query on the input can result in three possible actions:

1. The query can diverge. The element represented by I may be below b.

2. The query can answer true. In this case, the ideal I contains the finite element b.

3. The query can answer false along with a *witness* $c \in I$ that is inconsistent with b.

These possible responses to queries govern the way a continuous functions evaluates an input. The computation is performed by incrementally querying the input ideals in parallel for finite elements and generating the finite elements as soon as enough information about the input is gathered. Under this model a continuous function enumerates the elements in the ideal corresponding to the answer. At any point in the computation, the approximate answer is an ideal contained in the final ideal.

Each function under this model represents a method of querying the input and generating the output. It is easy to see that each query is continuous from the input domain to the boolean domain. Since each element is represented as the ideal corresponding to that element, the partial recursive enumeration of the ideal supplies the information about the element. If a function f yields an output b for an infinite element a, then there should be a *finite* element c that is in the ideal corresponding to a to prompt the output b. Hence, the function f should output b for the finite element c.

Unfortunately, this model of computation admits deterministic parallel operations. Consider the example of *por*:

Example 3.1 The function *por* over the booleans generates tt if at least one of the elements is tt and ff if both the elements are ff. It diverges if both the elements are $\perp$. When an application por(x,y) is evaluated, the function por has to examine both input arguments simultaneously to avoid diverging on one of the inputs $(\perp, \mathsf{tt})$ and $(\mathsf{tt}, \perp)$. Consequently, *por* must issue queries $(\mathsf{tt}, \perp)$ and $(\perp, \mathsf{tt})$ in parallel. If either of them is satisfied, the function will produce tt. If the probes return false, the input must be (ff, ff); por produces ff as the answer. ■

To restrict the continuous function space to sequential functions, we try to eliminate the parallel operations from the intuitive model. We progressively refine that attempt to produce the concept of sequential computation.

3.1.2 Sequential Computation – An Informal Description

Since the informal model contains the deterministic parallel operations, which clearly are not sequential, we must eliminate them to produce a model of sequential computation. An obvious modification is to change the type of queries and the possible actions.

Consider the example of *por*. The function queries for two finite elements simultaneously. It might appear that by restricting the queries to be performed sequentially, we can eliminate *por*. However, *por* can be written in the following manner with only a sequence of queries:

> **If the element (ff, ff) approximates the input, generate ff else generate tt.**

In the preceding example, a query is being made for both parts of a pair. Such a query can be decomposed into multiple queries. For example, the previous example issues a query for the first part of the pair and the second part of the pair, which are two queries. A negative response to such a compound query can reveal the presence of one of the elements inconsistent with the query. Thus a negative response to a compound query answers two queries done in parallel. If we want to restrict ourselves to asking queries in a sequence, we must avoid compound queries. What we need are "atomic" queries, that cannot be broken down into further queries.

Atomic Queries – Prime Elements: Since finite elements may correspond to compound queries, we must identify the elements that correspond to the atomic queries. For this purpose, we identify a subset of finite elements, called *prime elements*. Intuitively, prime elements correspond to atomic queries—they cannot be described using other elements.

Definition 3.1. (*Prime Element*) A finite element b is a prime iff whenever it is a least upper bound of a set B then $b \in B$. ∎

To define sequential computation, we should formulate Domain Theory in terms of prime elements. Almost all conventional domains (consistently complete, ω-algebraic cpo's) encountered in computer science applications can be described using prime elements. In the domain $\mathbf{N}_\perp$, all the numbers are prime elements. In the domain $\mathsf{P}\omega$, the powerset of integers, all the singleton sets are prime elements. The domain T^ω also contains prime elements; all the elements that have only one non-$\perp$ in the ω product are prime elements. Domains such as ∇[CD88] and $\mathcal{U}$[Sco82] cannot be described using the prime elements.

In the domains that can be described using prime elements, the consistency relation on the finite elements can be transformed to the consistency relation on sets of prime elements. Similarly, least upper bounds are defined on sets of prime elements. Most domains described using finite elements can be described using prime elements.

Sequential Functions: The description of domains using the prime elements leads to an improved definition of sequential functions: any function that performs the computation by a sequence of prime queries is a sequential function. At any stage in a computation, the function can issue only one query for a single incremental prime element in the input ideal. That is, the query element must be a prime and it must lie immediately above the information collected so far. If the function needs to test the presence of a finite element, it can do so by issuing queries for all the approximating prime elements in *some sequence*. The response to each query can reveal the presence of at most a single prime element.

Unfortunately, the preceding model of computation does not quite work in the context of functions as data. The function space determined by the preceding model cannot be described using prime elements. This function space construction does not yield a *cartesian closed category* [BC82], making it unsuitable to describe semantics of higher-order programming languages based on the typed λ-calculus. In short, it is impossible to treat functions as data under this framework.

3.1.3 Observably Sequential Functions – An Informal Description

The set of sequential functions, as described above, lack an essential capability. Given a sequential function f, there is no way to determine the evaluation order of f by inspecting the graph of f. This situation is in contrast to most "real world" programming languages. Most programming languages have control operations that reveal the evaluation strategy. Some examples are errors, call/cc, and print. In the presence of such operators, programmers can observe the order of evaluation.

Example 3.2 Consider the addition function that adds two numbers. There can be two implementations, with different evaluation sequences as follows:

```
left-add (x,y) = evaluate x;
evaluate y;
add x and y.

right-add (x,y) = evaluate y;
evaluate x;
add x and y.
```

if the language has a facility like abort which causes any evaluation to abort with its argument as the final answer, then we can differentiate those two functions. On the input pair (abort(0), abort(1)) the function `left-add` generates 0 while `right-add` generates 1. ■

In addition, if the language contains constructs such as *catch* and *throw* [CF92], *programs* can observe the sequence of evaluation. By introducing such *sequentializers* in the domain, we can augment the graph of the function by the behavior of the function on these sequentializers. The same sequential function may be augmented using several different evaluation orders, producing different *observably sequential "functions"*. However, such constructs force functions to contain intensional meaning, thus abandoning the order-extensionality[Cur93]. By adding suitable "error" elements we can recover the order-extensionality [CF92].

Error Elements: As a modification to the continuous function model, the *OS*-function model uses prime queries to gather information about the input ideal. A further modification is that the model recognizes the possibility of errors in the querying process, and propagates such errors to the output of the function. A function that produces an error e element in the output as result of encountering error e in the input is called an *error sensitive function*.

To accommodate the errors as possible answers to a query, we include error generating primes or error primes in the domain. If a query for a prime element encounters a corresponding error prime element, it responds an appropriate error prime element.

In the operational model, a query for a prime element b is equivalent to query for any of the prime elements that are inconsistent with b and are above the *lub* of elements gathered from the input so far. (Here, a query is expected to return a "true" answer, or a "false" answer with a witness). Because of this symmetry, we can divide all the prime elements above a finite element into equivalent sets called C-sets, short for (conflict sets). All the prime elements in a C-set are inconsistent with each other. Thus, a query for some element is equivalent to a query for another element in the same C-set. If every C-set contains error elements, then the witness for an error response would be the error element.

Observably Sequential Functions: The observably sequential functions or OS-functions use the structure of the C-sets to perform computations. The underlying model of computation can be intuitively described as follows. A query for a prime element p in an element b can generate one of the following results:

1. **Divergence:** The prime element p is not present in b.
2. **Presence of** p
3. **Absence of** p: Let C_p be the C-set containing p. We can further analyze this case as follows:
   (a) **Witness:** The element b contains a prime element from the C_p different from p.
   (b) **Error:** The element b contains an error prime from C_p, i.e, the querying process generates an error.

We can perform sequential computations involving functions as data, if the functions are built from the preceding model. Since the evaluation strategy is manifest in the graph of the function, we can make decisions based on evaluation order information too. We illustrate this point with an example.

Example 3.3 Let B be the boolean domain and B_e be the error enriched boolean domain. Consider the functions `left-or`, `right-or` defined over B_e as follows:

```
left-or (x,y)= if x then tt else y.
right-or(x,y)= if y then tt else x.
```

Since these two functions have different graphs in the boolean domain with error primes, we can show that `left-or` $\rightarrow$ tt $\sqcup$ `right-or` $\rightarrow$ ff is an OS-function.

At first glance it might look impossible, since such a function cannot be continuous in the Scott model of continuous function. However, the preceding two `or` functions are continuous and do not have a least upper bound.

In $B \times B \rightarrow_c B$ there are versions of left-or and right-or, called `left-or'` and `right-or'` respectively, that are not error-sensitive. Since these two functions both approximate por, there is no function `left-or'` $\rightarrow \mathsf{tt} \sqcup$ `right-or'` $\rightarrow \mathsf{ff}$. The functions `left-or'` and `right-or'` are not error sensitive and do not belong to $B_e \rightarrow_{es} B_e$, the error sensitive function space. ∎

To summarize, OS-functions are a subset of continuous that generate an error e when they encounter an error e in the evaluation of the input. Since these function require errors, they can only be defined on a special class of domains called OS-domains.

3.2 OS-domains

In this section, we define the class of observably sequential domains (OS-domains). These domains have the structure required to support observably sequential computation. These domains are similar to concrete domains [KP93]. Unlike concrete domains, these domains are constructed from *prime basis*. Besides, these domains contain error-elements that are essential to our formulation of domain theory.

Traditionally a domain is an *ω-algebraic cpo*, *i.e.*, a partial order with certain topological constraints. An alternative is to construct the domain from a basis satisfying the necessary criteria. In the following subsection, we describe the process of domain construction from its finitary basis.

3.2.1 Scott-domains

In Sect. 2.1, we defined domains and finite elements. In this section we define a *finitary basis*, a partial order, and from that partial order we construct a domain. Most of the terms we use are defined in Sec. 2.1.

Definition 3.2. (*Finitary Basis*) A partial order $\langle B, \sqsubseteq \rangle$ is a finitary basis iff B is countable and every finite bounded subset of B, including the empty set, has a least upper bound in B. ∎

Each finitary basis defines a domain. The elements of the finitary basis are the finite elements of the domain. The domain may have more elements than the finitary basis; these additional elements are infinite elements. We construct the domain from a finitary basis through the "ideal-completion" process. All of these definitions and proofs are from standard text books in domain theory [Gun92] and other papers [Sco76, Plo78, Plo77, CD88].

Definition 3.3. (*Ideal, Principal Ideal*) Let $\langle S, \sqsubseteq \rangle$ be a partial order. A non-empty set $R \subseteq S$ is an ideal, if

- R is directed, and
- R is downward closed, *i.e.*, if $r \in R$ then every $s \sqsubseteq r$ is in R.

An ideal R is principal if there is an element $s \in S$ such that $R = \{s' \in S \mid s' \sqsubseteq s\}$. ∎

Clearly, each element of the partial order has a corresponding principal ideal. However, there are ideals that are not principal.

Example 3.4 Consider the partial order $\langle N, \sqsubseteq_m \rangle$ described in the previous examples. The ideal corresponding to each natural number is the set of its divisors. The infinite ideals, such as the set of all even numbers, are not principal ideals, as they do not have a corresponding natural number. ∎

Given any bounded set R we can form the minimum ideal containing R by first least upper bound closing and then downward closing. We call such a set the "ideal closure" of R. Thus, principal ideals are ideal closures of finite elements.

We construct the domain from a finitary basis as follows.

Definition 3.4. (*Domain*) A domain $(D, \subseteq)$, often just denoted as D, is a partial order generated from the finitary basis $\langle B, \sqsubseteq \rangle$ such that the elements of D are ideals over $\langle B, \sqsubseteq \rangle$. ∎

It is clear that only one domain can be generated from a finitary basis. If two finitary bases are order-isomorphic, they produce order-isomorphic domains.

As stated earlier, each element in the finitary basis has a corresponding principal ideal. In a domain we can identify the set of finite elements as follows. An element a is a finite element iff $a \sqsubseteq \sqcup R$ where R is a chain then $a \sqsubseteq r \in R$.

Lemma 3.5 In a domain D constructed from the partial order $\langle B, \sqsubseteq\rangle$, finite elements are precisely the principal ideals.

Proof.

Part I: *A principal ideal is a finite element.* Let I_a be a principal ideal corresponding to a finite element $a \in B$, *i.e.*, $I_a = \{b \in B \mid b \sqsubseteq a\}$. Let $I_a \subseteq \bigsqcup R$, where R is a chain. We must show that there is an ideal R_1 such that $I_a \sqsubseteq R_1 \in R$. Since $I_a \subseteq \bigsqcup R$, $a \in \bigsqcup R$. If a is included in R, then there must be an ideal $R_1 \in R$ containing a. Therefore $I_a \subseteq R_1$.

Part II: *If I is a finite element, then it is a principal ideal.* Assume $I = \{a_1, a_2 \ldots\}$. Now construct A_i as the principal ideal corresponding to $\bigsqcup_{j \leq i} a_j$. By construction, $(i) A_i$ are principal ideals, (ii) they form a chain, (iii) Since I is directed, for all i, $A_i \subseteq I$, and (iv) $I \subseteq \bigsqcup A_i$. Since I is a finite element, $I \subseteq A_j$ for some j. Since $A_j \subseteq I$, $I = A_j$. Therefore I is a principal ideal.

$\square$

Now, we will prove that the partial order we constructed is a Scott-domain, *i.e.*, it has the required properties.

Lemma 3.6 The domain D constructed from a finitary basis $\langle B, \sqsubseteq\rangle$ has the following properties.

consistently complete: Every pair of bounded elements has a least upper bound.

ω-algebraic: Every element is the least upper bound of its finite approximations, and the number of finite elements are countable.

cpo: The domain is a complete partial order, or a cpo, that is,

- there is a least element denoted $\perp$,
- every bounded set R has a least upper bound denoted $\bigsqcup R$.

Proof. **Consistently Complete:** If a and b are two elements in D that are bounded, then $a \cup b$ is a bounded set of finite elements. By performing ideal closure operation on $a \cup b$, we get the minimal ideal that dominates a and b, hence it is $a \sqcup b$. Therefore the domain is consistently complete.

ω-algebraic: Since the number of finite elements in D is countable, the finite approximations to any element is countable. Also, let A be an element in the domain. Let $a_1, a_2, a_3 \ldots$ be the elements of the ideal A. Just as in the proof of lemma 3.5, we can construct the principal ideals A_i by the ideal closure of $\{a_i\}$. Then $A = \bigsqcup A_i$ and A_i are finite elements by the lemma 3.5. Hence A is the least upper bound of some of its finite approximations, hence all of its finite approximations.

cpo: Clearly the ideal containing only the $\perp$ is the least element in the domain. Every bounded set R has a least upper bound, which can be constructed by taking the union of all the ideals and performing the ideal-closure operation. Such an ideal can be constructed because the union of all ideals in R is bounded. □

Theorem 3.7 A partial order $(D, \sqsubseteq)$ is a domain iff its finite elements form a finitary basis under $\sqsubseteq$.

Proof.

Part I : *The finite elements of D form a finitary basis, if $(D, \sqsubseteq)$ is a domain.* If a and b are finite elements, then $a \sqcup b$ is also a finite element. Hence, lub of finite bounded subset of finite elements is a finite element. Also, since $\perp \in D$, $\perp$ can be designated as lub of an empty set. Since the number of finite approximations is countable, the finite elements are countable.

Part II: *If the finite elements form a finitary basis, then $(D, \sqsubseteq)$ is a domain.* We can show that there is an isomorphism between the domain D and the domain generated by ideal completion from its finite elements. Each element of D can be represented as lub of all its finite approximations.

□

Given any partial order $\langle S, \sqsubseteq \rangle$, we can find out if it is a domain by verifying that it is a consistently complete, ω-algebraic cpo. However, an easier alternative is to identify the finite elements of S and verify that they form a finitary basis under the ordering $\sqsubseteq$. If the ideal completion of such finitary basis is order-isomorphic to $\langle S, \sqsubseteq \rangle$, then $\langle S, \sqsubseteq \rangle$ is a domain.

3.2.2 OS-domains

OS-domains are Scott-domains with special structure: they can be generated from a subset of finite elements called the *prime elements*. The topological constraints governing OS-domains are formulated as constraints on the prime elements. In the next few definitions we present *prime basis* which we use to construct OS-domains. Winskel [Win80] proposed a similar approach to define the domains in his thesis, but he did not focus on sequential domains.

Prime Basis

A prime basis is not merely a partial order; it is is a partial order enhanced with a partitioning. In addition, such a partial order must satisfy some more conditions to be a prime basis.

Definition 3.8. (*Partitioned Partial Order*) A triple $\mathcal{P} = (P, \leq, C)$ is called *partitioned partial order* or *ppo* iff

[PO] $(P, \leq)$ is a countable partial order.

[C$_1$] C is an equivalence relation on P. P can be partitioned into $Q_1, Q_2 \ldots$ by C.

▮

Any partial order can be partitioned to form a *ppo*. A partitioning of the partial order is called a C-set. We refer to C-sets by capital letters such as Q and R. We augment these names by superscripts and subscripts. Superscripts indicate which *ppo* they are from and subscripts differentiate the C-sets within the same *ppo*.

We say that p and q are in *conflict* with each other when $(p,q) \in C$.

Notation: Depending on convenience, we specify C either as an equivalence relation or a partitioning of P. That is, we write $(p,q) \in C$ if we specify C as an equivalence relation, or $Q \in C$, when we specify C as a set of C-sets. Also, we use C as a function that maps the elements of the partial order into their C-sets. That is, $C(p)$ is the C-set containing p.

Definition 3.9. (*(Error-rich) Prime Basis*) A ppo $(P, \leq, C)$ is a *prime basis* iff

[UP] Each element of the partial order $(P, \leq)$ has at most one predecessor, *i.e.*,

$$\forall p, r_1, r_2 \in P : (r_1 \prec p \wedge r_2 \prec p) \Rightarrow r_1 = r_2$$

[F] Each element of P is approximated by only a finite number of elements.

[C$_2$] Elements in the same C-set have the same set of predecessors. *i.e.*,

$$\forall p, q, r \in P : (p,q) \in C \Rightarrow (r \prec p \Rightarrow r \prec q)$$

(**[E]**) A prime basis $(P, \leq, C)$ is *error-rich* iff each C-set $Q \in C$ has two designated elements error_1^Q, error_2^Q that are maximal in $(P, \leq)$. In addition, every C-set Q must contain at least one element other than error_1^Q, error_2^Q.

∎

All properties except **[E]** are topological constraints on the given ppo. The designation of error elements is given as a part of specification of an error-rich prime basis. The property **[E]** can be thought of as an oracle on the prime elements that can recognize the error elements.

Since each C-set has only one predecessor by property **[UP]**, we can extend the familiar notions of covering, preceding to the C-sets and prime elements too.

Definition 3.10. (*Coverage, Direction, Fringe, Distance*) Let $(P, \leq, C)$ be a prime basis. A C-set Q *covers* a *prime element* $p \in P$ (written $p \prec Q$) iff there is a $q \in Q$ such that $p \prec q$ (implying that $p \prec r$ for all $r \in Q$).

The *direction* relation is the transitive generalization of the *coverage* relation between primes and C-sets. More precisely, a prime element p has *direction* Q, where $Q \in C$ iff there exists $q \in Q : p \leq q$. We extend this notion to a relation on C-sets by saying that a C-set R is in the direction Q if there is a prime in R in the direction Q.

In conformance with the notation $p \leq q$, we write $p \leq Q$ when $p \leq q \in Q$.

A *fringe* for a finite element a is a set of primes that are maximal in a. That is,

$$S_a = \{p \in a \,|\, p \leq q \Rightarrow q \notin a\}$$

If p and q are primes in a prime-domain D such that $p \leq q$, the *distance from* p to q is the cardinality of the set of all the primes that approximate q, but not p. We extend this notion to C-sets by defining the distance between C-sets Q and R as the distance between any two primes $q \in Q$ and $r \in R$ such that $q \leq r$. ∎

Construction of OS-domain

From a given prime basis, we can generate a *prime domain* using a process similar to ideal completion. We form these ideals from prime elements as described below.

Definition 3.11. (*Prime Ideal, Principal-Prime Ideal*) Let $(P, \leq, C)$ be a prime basis. A set I is a prime ideal iff

- it is downward closed, and
- it does not have more than one element from the same C-set, *i.e.*,

$$p, q \in I \Rightarrow C(p) \neq C(q)$$

An ideal I is a principal-prime ideal iff there is an element $p \in P$ such that $I = \{p' \in P \mid p' \leq p\}$. We denote such as ideal I as I_p. ∎

Definition 3.12. (*Prime-Domain, OS-domain*) A *prime-domain* is a partial order $(D, \subseteq)$ generated from a prime basis $(P, \leq, C)$ where the elements of D are precisely the prime ideals.

The prime-domain generated by the prime basis $(P, \leq, C)$ is denoted $\mathbf{D}(P, \leq, C)$. If the prime basis $(P, \leq, C)$ is error-rich, then the prime-domain $\mathbf{D}(P, \leq, C)$ is an OS-domain. ∎

The domain generated by a prime basis is a Scott-domain, with additional constraints. We prove this fact by identifying a finitary basis that generates it as follows.

Lemma 3.13 The finitary basis for the domain $\mathbf{D}(P, \leq, C)$ is $(B, \subseteq)$ where B is the set of prime ideals with finite cardinality.

Proof. Let B be the set of finite prime ideals. Clearly, $(B, \subseteq)$ is a partial order.

Before showing that $(B, \subseteq)$ is a cpo, we observe that the lub of a set of prime ideals is the set union, if such union does not contain conflicting primes. That is, the set union produces a finite prime ideal.

The empty set is a finite ideal, representing $\perp$. Every finite bounded set of finite prime ideals R has a lub, which is simply the set union of all the members of R. Hence $(B, \subseteq)$ is a finitary basis.

We will show that the domain D' formed by ideal completion over B is order-isomorphic to $D = \mathbf{D}(P, \leq, C)$ as follows. An ideal R in the domain D' can be mapped to a prime-ideal containing only the set of prime elements in the ideal R. Similarly a prime ideal R' can be used to construct an ideal R as

$$r \in R \iff r \sqsubseteq \sqcup R'$$

It can easily be checked that this mapping respects the ordering relation in the domains. Hence these two domains are isomorphic. □

Identifying a Prime Domain

The preceding section describes how to build a prime domain. In this section we describe the conditions that a domain must meet to be a prime domain. To do so, we need to identify the prime elements of a domain.

Definition 3.14. (*Prime Element*) An element $a \in D$ is a prime element iff $a = \sqcup R$ for a bounded R implies $a \in R$. By convention, we take $\perp$ as a non-prime. ∎

Lemma 3.15 In a domain $D = \mathbf{D}(P, \leq, C)$, the prime elements are precisely the principal-prime ideals.

Proof.

Part I: *A principal-prime ideal is a prime element.* Let I_p be the principal-prime ideal corresponding to the element $p \in P$. If $I_p \subseteq \sqcup R$, we must show that $I_p \subseteq r \in R$. Since $I_p \subseteq \sqcup R$, we know that $p \in \sqcup R$, hence $p \in r \in R$. Since prime ideals are downward closed, $I_p \subseteq r \in R$.

Part II: *A prime element in D is a principal-prime ideal.* Let $d \in D$ be a prime element. Since d is a prime ideal, it must be downward closed containing no conflicting elements. Let the elements in d be $p_1, p_2 \ldots$. For each p_i, we can construct the corresponding principal-prime ideal I_{p_i}. Since the set of I_{p_i} is bounded and $d = \sqcup\{I_{p_i}\}$, we know that $d \subseteq I_{p_j}$ for some j, proving that d is indeed a principal-prime ideal.

□

By the preceding lemma, we identify a prime element with its principal-prime ideal. We use the principal-prime ideal and the corresponding prime element interchangeably.

Notation: We use the letters a, b, c for elements of OS-domains. We use p, q, r as prime elements of the underlying prime-basis. We write $a \sqsubseteq b$ iff $a \subseteq b$ when a

and b are expressed as sets and $a \sqcup b$ as the least upper bound of a and b. We write $p \in a$ when we mean that p is in the prime ideal representing a.

Now we identify some of the important properties of the domain D, which we use in the test for a prime domain later.

Lemma 3.16 A prime domain $D = \mathbf{D}(P, \leq, C)$ is a Scott-domain with the following properties:

[d] It is *distributive*, *i.e.*,

$$\forall a, b, c \in D : \ (a \sqcup b) \sqcap c = (a \sqcap c) \sqcup (b \sqcap c)$$

when the least upper bound of a and b exists.

[I] It is *finitely founded*, *i.e.*, every finite element has only finite number of finite approximations.

[$\mathbf{C}_3$] If two prime elements I_p, I_q of D are inconsistent, then there must be $I_{p'} \subseteq I_p$ and $I_{q'} \subseteq I_q$ such that $I_{p'}$ and $I_{q'}$ are inconsistent and have the same set of predecessors.

[$\mathbf{C}_4$ It is *coherent*, *i.e.*, every a pairwise consistent (every pair of elements are consistent) set has a lub.

Proof. (*i*) Since Intersection of two prime ideals is a prime ideal and union of two prime ideals is a prime ideal (provided the union does not contain any conflicting primes), distributivity trivially holds in the domain.

(*ii*) Since each finite element has only finite number of prime approximations, it must have only a finite number of finite approximations.

(*iii*) Let I_p, I_q be inconsistent. Then $I_p \cup I_q$ must contain two distinct elements from same C-set. Therefore, there must be a $p' \in I_p$ and $q' \in I_q$ such that $C(p') = C(q')$. The corresponding principal-prime ideals $I_{p'}$ and $I_{q'}$ approximate I_p and I_q respectively.

(iv) Let a set of prime ideals be inconsistent. Then there must two prime elements belonging to a C-set that are in the union of these prime ideals. Hence there must be two prime ideals that are inconsistent. Therefore, every inconsistent set has a pairwise inconsistency. $\square$

Now, we will identify the properties that a domain must satisfy to be a prime domain. First, we will identify the ppo of a given domain. If the set of primes forms a ppo and the domain satisfies certain properties, then the domain is a prime domain.

Definition 3.17. (*Competing Primes*) In a given Scott-domain $(D, \sqsubseteq)$, two primes are said to be competing iff $p \# q$ and p, q have same set of prime approximations. ∎

Lemma 3.18 Let $(D, \sqsubseteq)$ be a Scott-domain. Then D is a prime domain iff the following properties are satisfied:

Prime Basis: Let the set of primes of the domain be P. Also, let C be the competing relation between two primes. Then the triple $(P, \sqsubseteq, C)$ must be a ppo and form a prime basis.

Domain Properties: D must satisfy the properties **[d]**, **[I]**, **[C$_3$]**, and **[C$_4$]**.

Proof.

Part I: $(\Leftarrow)$ (i) *If D is a prime domain, then $(P, \sqsubseteq, C)$ is order-isomorphic to its prime basis.*

If D is a prime domain, then it must be formed from a prime basis $(P', \leq', C')$. By Lemma 3.15, P' is isomorphic to P, the set of principal-prime ideals of D. Therefore, for any element $p' \in P'$ there is a corresponding principal-prime ideal $I_{p'}$.

Also, if for any $p', q' \in P'$, if $p' \leq' q'$ then $I_{p'} \sqsubseteq I_{q'}$. Hence $(P', \leq')$ is order-isomorphic to $(P, \sqsubseteq)$.

Similarly, if $(p', q') \in C$, they cannot belong to a C-set; hence $I_{p'}$ and $I_{q'}$ do not have a least upper bound. If p' and q' has a predecessor r, then $I_{p'}$ and $I_{q'}$ have a predecessor $I_{r'}$. Also, if p', q' do not have a predecessor, $I_{p'}, I_{q'}$, do not have a predecessor.

Hence $(P, \sqsubseteq, C)$ is order-isomorphic to $(P', \leq', C')$, the prime basis for D. Hence D is isomorphic to $\mathbf{D}(P, \sqsubseteq, C)$.

(ii) Since D is a prime domain, by Lemma 3.16 it satisfies the properties **[d], [I]**,**[C$_3$]**, and **[C$_4$]**.

Part II: $(\Rightarrow)$ *If* $(P, \sqsubseteq, C)$ *forms a prime basis, and* D *satisfies the properties* **[d], [I]**, *and* **[C$_3$]** *then* $\mathbf{D}(P, \sqsubseteq, C)$ *is order isomorphic to* $(D, \sqsubseteq)$.

We establish the mapping between the finite elements of $D' = \mathbf{D}(P, \sqsubseteq, C)$ and the finite elements of $(D, \sqsubseteq)$ as follows. We show that this mapping is one-to-one and onto and preserves the order. Thus, we show that D' and D are order-isomorphic.

Any finite prime ideal of P (*i.e.*, a finite element in D') has a least upper bound by property **[C$_3$]**. So, we associate a finite prime ideal with its least upper bound.

(i) Distinct prime ideals represent distinct elements.

First of all, each finite prime ideal represents a finite element. If it does not, then there must be a set of prime elements that are inconsistent. By property **C$_4$** (coherence), there must be a pair of elements that are inconsistent, which cannot be the case by the definition of prime-ideal.

Assume that there are two prime ideals A and B with the same least upper bound a. It is easy to observe that neither A nor B is a principal prime ideal.

Let $p \in A$ and $p \notin B$. Since B must have more than one prime element, let $B = \{p_1, p_2, p_3 \ldots\}$. Since D is distributive, we have the equality:

$$(p_1 \sqcup p_2 \sqcup p_3 \ldots) \sqcap p = (p_1 \sqcap p) \sqcup (p_2 \sqcap p) \ldots$$

Since B is downward closed, the left hand side is equal to $a \sqcap p = p$, while the right hand side is strictly below p, since p is a prime element and each of $p_i \sqcap p$ is strictly below p. Therefore, we have a contradiction that D is not distributive. Hence, distinct prime ideals represent distinct elements.

(ii) Each finite element is the least upper bound of a finite prime ideal.

We will show this proposition by induction on the distance between $\perp$ and the finite element. Define height of an element a of a domain as follows. If $a = \perp$, then its height is 0. Otherwise, its height is one more than the maximum of heights of elements that are strictly below it. Obviously, in a finitely founded domain, height of every finite element is finite.

Induction Hypothesis: If the height of the finite element a is n, then it is the least upper bound of its prime approximations.

Base Case: $n = 0$. Then, since the set of prime approximations is empty, and $\perp$ is the lub of empty set, hence IH holds.

Inductive Case: Assume IH holds for $n = k$. For $n = k + 1$, we must show that $a = \sqcup A$ where A is the set of its prime approximations.

$Case(i)$: a is a prime element. Then, we know that A contains a and all the members of A approximate a. Hence A is a prime ideal, whose lub is a.

$Case(ii)$: a is not a prime element. Let A' be the set of all the finite approximations to A except a. By IH, every element of A is the lub of its prime approximations. By taking the union of all those prime approximations, we get a prime ideal with least upper bound a.

Hence, by induction, we show that any finite element is the least upper bound of its prime approximations.

□

We can also show that a given domain is a prime domain by constructing an isomorphic domain with some prime basis. From Lemma 3.18, we know that we must start from the prime elements of the domain to construct a prime basis.

Most domains used for the models of deterministic languages are prime-domains. The most notable exceptions are the domains generated by the continuous function construction. Consider the following examples.

1. The flat domain $\mathbf{T}_\perp$ of truth values $\{\mathsf{tt}, \mathsf{ff}\}$ is a prime-domain; the truth values $\{\mathsf{tt}, \mathsf{ff}\}$ are the prime elements. These elements are in conflict with each other.

2. The flat domain $\mathbf{N}_\perp$ of natural numbers is a prime-domain; every number is a prime element. All the numbers are in conflict with each other.

3. Scott's universal domain $\mathcal{U}$ is described as follows. The elements are trees over $\perp, \top$ under the equivalence relation $\langle \top, \top \rangle = \top$ and $\langle \perp, \perp \rangle = \perp$. That is each tree is collapsed to form canonical trees.

   The order relation $\sqsubseteq$ is defined as $\perp \sqsubseteq \top$ and $\langle a, b \rangle \sqsubseteq \langle a', b' \rangle$ iff $a \sqsubseteq a'$ and $b \sqsubseteq b'$.

   This domain is not a prime domain because it is not finitely founded. Any finite element has infinitely descending chain as follows. Any non-bottom tree must have a top; otherwise it collapses to $\perp$. Since,

   $$\top = \langle \perp, \top \rangle \sqcup \langle \top, \perp \rangle,$$

   any non-bottom tree is the lub of two trees obtained by replacing the $\top$ with $\langle \perp, \top \rangle$ and $\langle \top, \perp \rangle$ respectively. By repeating this process, we can obtain the infinite descending chain.

4. Recall that a function $f \in D_1 \rightarrow_c D_2$ iff $f(\bigsqcup d_i) = \bigsqcup f(d_i)$. Alternately, if the function is applied to a least upper bound of a chain, then the result is equal to

the least upper bound of the results obtained by applying the function to each of the elements of the chain.

The continuous function space $\mathbf{T}_\perp \rightarrow_c \mathbf{T}_\perp$ can be described by the prime elements which are the "one-step" functions:

$$(d \Rightarrow e)(x) \stackrel{df}{=} \begin{cases} e & \text{if } d \sqsubseteq x \\ \perp & \text{otherwise} \end{cases}$$

where $d, e \in \mathbf{T}_\perp$ and $e \neq \perp$. Thus, the prime elements are

$$\perp \Rightarrow \mathsf{tt}, \perp \Rightarrow \mathsf{ff}, \mathsf{tt} \Rightarrow \mathsf{tt}, \mathsf{tt} \Rightarrow \mathsf{ff}, \mathsf{ff} \Rightarrow \mathsf{tt}, \mathsf{ff} \Rightarrow \mathsf{tt}.$$

This domain is not a prime domain because the derived ppo of primes does not satisfy the [**UP**] property.

We can "weakly" generate the domain from the prime elements as follows. Define a relation # over the prime elements such that $p_1 \# p_2$ iff p_1 and p_2 are inconsistent in the original domain, which is $\mathbf{T}_\perp \rightarrow_c \mathbf{T}_\perp$ in this case. The domain can be formed as down-ward closed sets of prime elements such that no two elements are related by #. These elements are ordered by the subset relation.

5. The continuous function space $\mathbf{N}_\perp \rightarrow_c \mathbf{N}_\perp$ is not a prime-domain, but it is weakly generated by its prime elements, which are the one-step functions, as described in the preceding example. $\mathbf{N}_\perp \rightarrow \mathbf{N}_\perp$ is not a prime-domain, because the prime basis violates both property [**F**] and property [**C**]. The prime element $(\perp \Rightarrow 1)$ has infinitely many prime elements $(n \Rightarrow 1)$ $(n \in \mathbf{N})$ below it.

   In the sequel, we will use "one-step" functions as parts of prime functions over OS-domains. As we saw in the examples above, the prime elements of non-trivial continuous function spaces do not form a prime basis, because they fail property [**C**]. They also fail property [**F**] if the input space is infinite.

6. A simple example of an observably sequential domain (a prime-domain with designated error elements) is $\mathbf{N}_{\perp}^{E}$, the flat domain of natural numbers plus two error elements, error_1 and error_2. It is generated by the prime basis $(\mathbf{N}^E, =, C)$ where

$$\begin{aligned} \mathbf{N}^E &= \mathbf{N} \cup \{\mathsf{error}_1, \mathsf{error}_2\} \\ C &= \{\mathbf{N}^E\} \end{aligned}$$

7. The domain of trees over booleans and errors is an example of an OS-domain. That is, $(B_t, \sqsubseteq)$ where,

$$B_t ::= \perp \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{error}_1 \mid \mathsf{error}_2 \mid \langle B_t, B_t \rangle$$

and,

$$\begin{array}{lrcl} \forall x \in B_t & \perp & \sqsubseteq & x \\ \forall x \in B_t & x & \sqsubseteq & x \\ \forall x_1, y_1, x_2, y_2 \in B_t & \langle x_1, y_1 \rangle & \sqsubseteq & \langle x_2, y_2 \rangle \text{ iff } x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2 \end{array}$$

The prime elements of this domain are the special trees, called the *paths*, where a path is defined as

$$path ::= \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{error}_1 \mid \mathsf{error}_2 \mid \langle \perp, \perp \rangle \mid \langle path, \perp \rangle \mid \langle \perp, path \rangle$$

We use this domain for further examples in the rest of the chapter.

8. A more interesting example of observably sequential domain is the domain of functions generated by error-rich prime basis: $(P_{os}, \leq_{os}, C_{os})$. P_{os} consists of three disjoint subsets of continuous functions from $\mathbf{N}_{\perp}^{E}$ to $\mathbf{N}_{\perp}^{E}$:

Constant primes generate outputs independent of their inputs. Since a function of this form ignores its inputs, it never returns an error value unless its constant output is an error value. For every $e \in \mathbf{N}^E$, there exists one constant prime:

$$g_e = (\bot \Rightarrow e).$$

Strict one-step primes output designated elements if their inputs contain enough information. If the input is an error value error_i, a strict one-step prime returns error_i. For every pair $d \in \mathbf{N}, e \in \mathbf{N}^E$, there is a strict, one-step prime:

$$f_{d,e} = (d \Rightarrow e) \sqcup (\mathsf{error}_1 \Rightarrow \mathsf{error}_1) \sqcup (\mathsf{error}_2 \Rightarrow \mathsf{error}_2).$$

The diverging prime s inspects its inputs and then diverges. If the input is an error value e, the function s returns e:

$$s = (\bot \Rightarrow \bot) \sqcup (\mathsf{error}_1 \Rightarrow \mathsf{error}_1) \sqcup (\mathsf{error}_2 \Rightarrow \mathsf{error}_2).$$

In summary, $P_{os} = \{g_e, f_{d,e}, s \mid d \in \mathbf{N}, e \in \mathbf{N}^E\}$. The approximation ordering on P_{os} is the usual pointwise approximation ordering on functions. The conflict partitioning for P_{os} is:

$$C = \{B, C_k \mid k \in \mathbf{N}\}$$

where

$$\begin{aligned} B &\stackrel{df}{=} \{s, g_e \mid e \in \mathbf{N}^E\} \\ C_k &\stackrel{df}{=} \{f_{k,e} \mid e \in \mathbf{N}^E\} \end{aligned}$$

The designated error elements in B and C_k are g_{error_i} and f_{k,error_i} for all k, respectively.

Technically, each element in the prime domain $\mathbf{D}(P_{os}, \leq_{os}, C_{os})$ is a set of functions rather than a function. However, we can identify each such set of functions with its least upper bound under the usual pointwise ordering on functions. Since every function in P_{os} is continuous, these bounding functions are also continuous.

A simple example is the set $\{s, f_{k,k} \mid k \in \mathbf{N}\} \in \mathbf{D}(P_{os}, \leq_{os}, C_{os})$, which has the identity function as its least upper bound. Similarly, the set $\{s, f_{k,k+1} \mid k \in \mathbf{N}\}$ corresponds to the successor function.

Most domains can be enriched with errors to form *OS*-domains, except the domains of continuous functions. Since we do not use continuous functions to construct higher-order types, we do not encounter the non-*OS*-domains in our formulation of domain theory.

In the rest of the section we describe the function construction between the two *OS*-domains such that it forms an *OS*-domain. Before describing a general *OS*-function construction, we need to introduce the concept of "coverage".

A C-set Q covers a finite element a iff for any element $q \in Q$, $\{q\} \cup a$ is an element of the domain and $a \prec \{q\} \cup a$.

Hence, the elements of a C-set Q covering a are mutually exclusive bits of information that can be incrementally added to the set of information a. A finite element is typically covered by several different C-sets.

3.2.3 *OS*-Function Spaces

The set of *OS*-functions between two *OS*-domains A and B is a subset of the set of continuous functions $A \rightarrow_c B$. These functions propagate error values returned by *inspected* inputs. Since inputs are inspected only when they are "needed", we must formalize the concept of "need" to understand the semantics of error propagation. The notion of a sequentiality index captures the informal idea of "need" in a general context. The following definition adapts this notion to *OS*-domains.

Definition 3.19. (*Sequentiality Index*) Let $D_1 = \mathbf{D}(P_1, \leq_1, C_1), D_2 = \mathbf{D}(P_2, \leq_2, C_2)$ be *OS*-domains. For a continuous function $f : D_1 \rightarrow_c D_2$, a *sequentiality index* for finite input $a \in D_1$ and output C-set Q^2 covering $f(a)$ is a C-set R^1 covering a such that for all $x \sqsupset a$, $(f(x) \cap Q^2) \neq \emptyset$ implies $(x \cap R^1) \neq \emptyset$.

We refer to the set of sequentiality index of f at a for Q^2 as $si_f(a, Q^2)$. ∎

The presence of the error elements in the domain force a function to indicate which C-set above a is explored first. If R^1 is a sequentiality index of f at a and Q^2, then f explores R^1 while generating an output at Q^2. That is, f explores the input x which is above a for any element $r \in R^1$. If f propagates errors, then it must generate the output prime $\mathsf{error}_i^{Q^2}$ if r is $\mathsf{error}_i^{R^1}$. Hence, the graph of such a function implicitly contains its evaluation strategy. Since the graph of the function is available for exploration by functions such as *apply* and *catch*[CF92], the function is called observably sequential function or OS-function for short.

Definition 3.20. (*Observably Sequential Function*) Let D_1 and D_2 be OS-domains, generated from the bases $(P_1, \leq_1, C_1)$ and $(P_2, \leq_2, C_2)$, respectively. A continuous function $f : D_1 \rightarrow_c D_2$ is an OS-function iff it is

sequential: for every pair (a, Q^2), where Q^2 covers $f(a)$, there is a sequentiality index R^1 if $q \in f(x)$ for some $q \in Q^2$ and $x \sqsupset a$; and

error sensitive: if R^1 is the sequentiality index of f for the input a and the output C-set Q^2, $\mathsf{error}_1^{Q^2} \in (f(a \cup \{\mathsf{error}_1^{R^1}\}))$ and $\mathsf{error}_2^{Q^2} \in f(a \cup \{\mathsf{error}_2^{R^1}\})$.

We use $D_1 \rightarrow_{os} D_2$ to denote the set of the OS-functions between two OS-domains D_1, D_2. ∎

The following lemma shows that there can be at most one sequentiality index for any finite input element and output C-set.

Lemma 3.21 Let $f \in D_1 \rightarrow_{os} D_2$ and $a \in D_1$ and Q is a C-set covering $f(a)$. Then the set of sequentiality indices $si_f(a, Q)$ is either empty or consist of only one C-set$R$, where R covers a.

Proof. Let R be in $si_f(a, Q)$. Then

$$\forall i \in \{1, 2\} : \mathsf{error}_i^Q \in f(a \cup \mathsf{error}_i^R)$$

If there is an R' in $si_f(a, Q)$ then

$$\forall i \in \{1, 2\} : \mathsf{error}_i^Q \in f(a \cup \mathsf{error}_i^{R'}).$$

If we consider the function f on the input $a \cup \mathsf{error}_1^R \cup \mathsf{error}_2^{R'}$. Since the result must contain two conflicting elements error_1^Q and error_2^Q, we cannot have more than one sequentiality index.

$si_f(a, Q)$ can be empty if f is a constant function at a. □

Thus, an OS-function $f : D_1 \rightarrow_{os} D_2$ has at most one sequentiality index for a given finite input $a \in D_1$ and C-set Q^2 covering $f(a)$. Therefore, for OS-functions, it makes sense to introduce the function si where $si_f(a, Q)$ is the sequentiality index of f for a and Q.

Now, we must prove that the set of OS-functions are OS-domains. It is essential that they be Scott-domains for performing computation on function spaces. To perform observably sequential computation, these domains must form OS-domains. We will show this property in the following lemmas. First, we identify some crucial properties of OS-functions that help us determine the prime basis for the OS-function space.

Lemma 3.22 The least upper bound of two OS-functions in the continuous function space is an OS-function.

Proof. Let f and g be two OS-functions in $D_1 \rightarrow_{os} D_2$ with least upper bound h in $D_1 \rightarrow_c D_2$. Let (a, Q) be a pair where Q covers $h(a)$. If $h(x) \cap Q \neq \emptyset$ and $x \supset a$, we need to show that h has a unique sequentiality index R^1 at (a, Q). We also need to show that $\mathsf{error}_i^Q \in h(a \cup \mathsf{error}_i^{R^1})$ for $i = 1, 2$.

At least one of $si_f(a, Q), si_g(a, Q)$ is defined:

Since Q covers $h(a)$, it must cover at least one of $f(a)$ and $g(a)$. There are three possible cases with respect to sequentiality indices for f and g at (a, Q).

Only one of them is defined. In this case, assume without loss of generality that sequentiality index is defined for f.

Both of them are defined. Let them be R_f, R_g respectively. We show that these must be equal. Otherwise, the following hold true:

$$\mathsf{error}_a^Q \in f(a \cup \mathsf{error}_i^{R_f})$$

$$\mathsf{error}_a^Q \in g(a \cup \mathsf{error}_i^{R_g})$$

Therefore, $g(a \cup \mathsf{error}_1^{R_f} \cup \mathsf{error}_2^{R_g})$ must contain two conflicting elements if $R_f \neq R_g$. Therefore, they must be equal.

None of them are defined. This case cannot happen because there exists x above a such that $h(x) \cap Q \neq \emptyset$. Therefore, either $f(x) \cap Q \neq \emptyset$ or $g(x) \cap Q \neq \emptyset$.

In either case, $R_f = si_h(a, Q)$ because $h(a \cup \mathsf{error}_1^{R_f}) \supseteq f(a \cup \mathsf{error}_i^{R_f})$. Since f is observably sequential, $\mathsf{error}_i^Q \in h(a \cup \mathsf{error}_1^{R_f})$. $\square$

However, the greatest lower bound of a two OS-functions in the continuous function space may not be an OS-function. Consider the following example.

Example 3.5 Let f and g be OS-functions in $B_t \rightarrow_{os} B_t$ described by the skeleton graphs:

$$f = \begin{cases} \perp & \rightarrow & \perp \\ \mathsf{error}_i & \rightarrow & \mathsf{error}_i \\ \langle \perp, \perp \rangle & \rightarrow & \perp \\ \langle \mathsf{error}_i, \perp \rangle & \rightarrow & \mathsf{error}_i \\ \langle \mathsf{tt}, \perp \rangle & \rightarrow & \perp \\ \langle \mathsf{tt}, \mathsf{error}_i \rangle & \rightarrow & \mathsf{error}_i \\ \langle \mathsf{tt}, \mathsf{tt} \rangle & \rightarrow & \mathsf{tt} \end{cases}$$

$$g = \begin{cases} \perp & \to & \perp \\ \mathsf{error}_i & \to & \mathsf{error}_i \\ \langle \perp, \perp \rangle & \to & \perp \\ \langle \perp, \mathsf{error}_i \rangle & \to & \mathsf{error}_i \\ \langle \perp, \mathsf{tt} \rangle & \to & \perp \\ \langle \mathsf{error}_i, \mathsf{tt} \rangle & \to & \mathsf{error}_i \\ \langle \mathsf{tt}, \mathsf{tt} \rangle & \to & \mathsf{tt} \end{cases}$$

The functions f and g explore the pair differently: f evaluates the first part of the pair before the second part, while g does exactly the opposite. It can easily be verified that these two functions are OS-functions. Consider pointwise lub of f and g, say h:

$$h = \begin{cases} \perp & \to & \perp \\ \mathsf{error}_i & \to & \mathsf{error}_i \\ \langle \perp, \perp \rangle & \to & \perp \\ \langle \perp, \mathsf{tt} \rangle & \to & \perp \\ \langle \mathsf{tt}, \perp \rangle & \to & \perp \\ \langle \mathsf{tt}, \mathsf{tt} \rangle & \to & \mathsf{tt} \end{cases}$$

The function h is not an OS-function, because it is not error-sensitive at $\langle \perp, \perp \rangle$. ∎

Notation: *Writing a function:* We denote the function with its graph whose minimal, continuous extension describes the function completely. Thus, h in the previous example can be written as $\{(\mathsf{error}_i, \mathsf{error}_i)(\langle \mathsf{tt}, \mathsf{tt} \rangle, \mathsf{tt})\}$.

Lemma 3.23 If $f : D_1 \to_{os} D_2$ and $g : D_2 \to_{os} D_3$, then $g \circ f$ is a function in $D_1 \to_{os} D_3$. where $(f \circ g)(x) = f(g(x))$.

Proof. Let $h = g \circ f$. Since f and g are continuous, $h \in D_3 \to_c D_2$. In addition, we must show that h is an OS-function.

Let $h(a)$ be covered by Q and $h(b) \cap Q \neq \emptyset$. We must show that $si_h(a, Q)$ is defined, and h is error-sensitive. Since $h(b) \cap Q \neq \emptyset$, $g(f(a)) \cap Q \neq \emptyset$. Therefore,

$Q^* = si_g(f(a), Q)$ is defined. Since Q^* covers $f(a)$ and $f(b) \cap Q^* =\neq \emptyset$, $si_f(a, Q^*) = Q'$. We can easily check that the sequentiality index of h at (a, Q) is Q'. If any $y \sqsupset a$ such that $h(y) \cap Q$ is not empty, then $f(y) \cap Q^*$ is not empty, hence $y \cap Q'$ is not empty.

To show that h is error-sensitive, consider $a' = a \sqcup \mathsf{error}_i^{R^3}$. Since $\mathsf{error}_i^{R^1} \in g(a')$, $\mathsf{error}_i^{Q^2} \in f(g(a'))$. Therefore, $\mathsf{error}_i^{Q^2} \in h(a')$. □

Lemma 3.24 The $\rightarrow_{os}$ function space is stable. That is, if $a_1 \sqcup a_2$ exists then $f(a_1 \sqcap a_2) = f(a_1) \sqcap f(a_2)$.

Proof. Let $a_3 = a_1 \sqcap a_2$. If $q \in f(a_1) \sqcap f(a_2)$, we will show that $q \in f(a_3)$ too. If not, then there must be a C-set Q in the direction of q that covers $f(a_3)$. Since $f(a_1)$ contains a prime in the C-set Q, the sequentiality index R at (a_3, Q) must exist where $R \cap a_3 = \emptyset$. Since that index is unique, a_1 and a_2 must contain a prime element from R. Because a_1 and a_2 are consistent, they must contain the same prime from the C-set R, *i.e.*, $(a_1 \sqcap a_2) \cap R \neq \emptyset$. But this inequality contradicts the assumption that $a_3 = a_1 \sqcap a_2$. Therefore, if $f(a_1) \sqcap f(a_2) \sqsubseteq f(a_3)$. We already have the approximation in the other direction, *i.e.*, $f(a_3) \sqsubseteq f(a_1) \sqcap f(a_2)$, hence $f(a_3) = f(a_1) \sqcap f(a_2)$. □

As a consequence, we can define a set of mutually inconsistent set of finite elements called *thresholds*, for a given prime element.

Definition 3.25. (*Thresholds*) Let $D_i = \mathbf{D}(P_i, \leq_i, C_i)$ for $i = 1, 2$, and $f \in D_1 \rightarrow_{os} D_2$. A set S_q of finite elements in D_1 is said to be a set of *thresholds* for q iff

$$\forall x \in D_1 : (q \in f(x) \iff \exists a \in S_q : a \subseteq x)$$

∎

It is easy to see that by Lemma 3.24, the elements of the threshold set must be mutually inconsistent. For example, if a and b in the threshold set for p are inconsistent, then $f(a \sqcap b)$ must contain p. Therefore $a \sqcap b$ must also belong to the threshold set, which is not possible by definition of thresholds.

It is also easy to observe that if a threshold value contains an error prime, the corresponding output prime must be an error element. And, threshold value can contain at most one error prime.

To prove that $D_1 \rightarrow_{os} D_2$ is an OS-domain, we need to identify the prime elements of the space, and provide a partitioning.

Definition 3.26. (*The ppo* $\mathcal{P}_{\rightarrow} = (P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$) Let $\mathbf{D}(P_i, \leq_i, C_i)$ be OS-domains D_i, for $i = 1, 2$. We define a prime basis $(P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$ as follows.

- We define the set $P_{\rightarrow}$, a subset of $D_1 \rightarrow_{os} D_2$, consisting of special functions in $D_1 \rightarrow_{os} D_2$ that map finite elements of D_1 to prime elements of D_2. The set $P_{\rightarrow}$ contains two disjoint subsets:

  1. **Output Primes:** For each $a \in D_1$ and $p \in P_2$, we define the set of output primes as follows:

     $$a \xrightarrow{o} p = \{f \in D_1 \rightarrow_{os} D_2 \mid f(a) = p \wedge (\nexists g \sqsubset f : g(a) = p\}$$

     The label of a function $f \in a \xrightarrow{o} p$ is $\langle a, Q^2, F \rangle$ where $Q^2 \in C_2$ is the C-set containing p and $F = \{g \in D_1 \rightarrow_{os} D_2 \,|\, g \subset f\}$.

  2. **Schedule Primes:** For each $a \in D_1$, $R^1 \in C_1$ and $Q^2 \in C_2$ where $a \prec R$, we define the set of schedule primes as follows:

     $$a : R^1 \xrightarrow{s} Q^2 = \{f \in D_1 \rightarrow_{os} D_2 \,|\, si_f(a, Q^2) = R^1 \wedge \nexists g \sqsubset f : si_g(a, Q^2) = R^1\}$$

     The function $f \in a : R_1 \xrightarrow{s} Q^2$ has the C-set *label* $\langle a, Q^2, F \rangle$ where $F = \{g \in D_1 \rightarrow_{os} D_2 \,|\, g \subset f\}$.

- $\leq_{\rightarrow}$ is defined as the usual pointwise ordering.

- $C_{\rightarrow}$ is the partitioning of $P_{\rightarrow}$ determined by the inconsistency relation: p and q are inconsistent iff they have same C-set labels.

- The two designated error elements of a C-set with label $\langle a, Q, F'\rangle$ are $F' \cup \{(a, \mathsf{error}_1^Q)\}$ and $F' \cup \{(a, \mathsf{error}_2^Q)\}$.

∎

It is easy to see that $\mathcal{P}_{\rightarrow}$ is an error-rich ppo. We will show that it is a prime basis that generates the domain $D_1 \rightarrow_{os} D_2$.

Lemma 3.27 Let $D_1 = \mathbf{D}(P_1, \leq_1, C_1), D_2 = \mathbf{D}(P_2, \leq_2, C_2)$, and F be an OS-function in $D_1 \rightarrow_{os} D_2$. Let a be a threshold for $q \in Q$, a C-set in P_2. Then,

1. if a does not contain an error prime, then $\exists f \in a \overset{o}{\rightarrow} p : f \sqsubseteq F$.
2. if $a = a' \sqcup \mathsf{error}_1^R$ and $q = \mathsf{error}_1^Q$ or $a = a' \sqcup \mathsf{error}_2^Q$ and $q = \mathsf{error}_2^Q$, then $\exists f \in a' : R \overset{s}{\rightarrow} Q : f \sqsubseteq F$.

Proof.

Case 1: We define f as the minimal function in $a \overset{o}{\rightarrow} p$ satisfying the following criteria.

$$\begin{array}{rcll} f(y) & = & F(y) \sqcap q & \text{if } y \subseteq a \\ si_f(y, Q^2) & = & si_F(y, Q^2) & \text{if } Q^2 \text{ covers } F(y) \;\&\; Q^2 \text{ is in the direction of } Q \end{array}$$

It is easy to see that such as f is an OS-function with threshold a for p. The constraints do not take f outside the set $a \overset{o}{\rightarrow} p$.

Case 2: We define the schedule prime $f \in a' : Q \overset{s}{\rightarrow} R$ as the minimal function satisfying the following criteria.

$$\begin{array}{rcll} f(y) & = & F(y) \sqcap \mathsf{error}_1^Q & \text{if } y \subseteq a \\ si_f(y, Q^2) & = & si_F(y, Q^2) & \text{if } Q^2 \text{ covers } F(y) \text{ and } Q^2 \text{ is in the direction of } Q \end{array}$$

It is easy to verify that f is a schedule prime approximating F belonging to $a' : R \overset{s}{\rightarrow} Q$.

□

We further strengthen the preceding lemma by showing only one function from $a \xrightarrow{o} p$ and $a : R \xrightarrow{s} Q$ can approximate a function.

Lemma 3.28 Let f and g be the functions in $P_{\rightarrow}$. Then,

- if $f, g \in a \xrightarrow{o} p$ and $f \uparrow g$ then $f = g$; and
- if $f, g \in a : R^1 \xrightarrow{s} Q^2$ and $f \uparrow g$ then $f = g$.

Proof.

Part I: If $f \neq g$, then we must have a minimal $a_1 \in D_1$ such that $f(a_1) = p_1$ and $g(a_1) = p_2$ such that $p_1 \neq p_2$. We know such a minimal a_1 exists because D_1 satisfies property **[F]**. Without this property the set of elements where f and g could form an infinite descending chain without any minimal elements.

Case 1: a_1 does not contain an error. In this case, p_1 and p_2 are not error elements. Otherwise f and g cannot be prime functions. We also know that $p_1 \leq p$ and $p_2 \leq p$. Hence p_1 and p_2 must form a chain. With out loss of generality assume that $p_1 < p_2$. Let $p_1 \prec q \in Q$ such that $q \leq p_2$. Therefore, f must have a sequentiality index R at a_1 and Q. Therefore, $f(a_1 \sqcup \mathsf{error}_i^R) = \mathsf{error}_i^Q$. Since $f \uparrow g$, we must have $\mathsf{error}_i^Q \uparrow p_2$ for $i = 1, 2$. This, however, is impossible since p_2 can be consistent with only one of the errors as $p_2 \cap Q$ is non-empty. Notice that we require two designated error elements for this proof, since p_2 itself can be an error element.

Case 2: a_1 contains an error element. Then a_1 contains only one error element. Otherwise a_1 cannot be the minimal element where f and g differ on the input. Let $a_1 = a_2 \sqcup \mathsf{error}_1^R$, where $R \cap a = r$. If the inequality $f(a_2) \neq g(a_2)$ does not hold f and g have different sequentiality indices, which means f and g do not have a lub. If $f(a_2) \neq g(a_2)$, then a_2 is smaller than a_1 where f and g differ in the output, which contradicts the assumption. Hence, $f = g$.

Part II: Reducible to the previous case, because there are $f', g' \in (a \sqcup r) \stackrel{o}{\rightarrow} q$ for some $r \in R^1$ and $q \in Q^2$ such that $f \prec f'$ and $g \prec g'$.

$\square$

Now, we show that the ppo $(P_{\rightarrow}, \leq, C_{\rightarrow})$ is a prime basis as follows.

Theorem 3.29 The ppo $\mathcal{P}_{\rightarrow} = (P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$ is a prime basis.

Proof.

Part I: [UP] *and* **[F]** *properties.* Let f be a prime element in $a \stackrel{o}{\rightarrow} q$ with the C-set label $\langle a, Q^2, F' \rangle$. We prove that F' is a finite chain, that is, contains only a finite number of elements and they can be linearly ordered under $\leq_{\rightarrow}$.

Since q has a chain of prime elements approximating it, let $p_0 \prec p_1 \prec p_2 \ldots \prec p_n = q$. By Lemma 3.27, we know that there are output primes f_i approximating f such that $f_i \in a_i \stackrel{o}{\rightarrow} p_i$. By the same lemma we also know that $a_i \subseteq a_{i+1}$. By lemma 3.28, we know that $f_i \leq f_{i+1}$ also. Therefore, these output primes form a chain.

However, there are schedule primes approximating f. We must show that they are finite in number and they form a chain together with the output primes $\{f_i\}$.

We show that the schedule primes between f_i and f_{i+1} form a finite chain. Let $a_{i+1} - a_i = A$, a set of primes. That is, each prime in A is only in a_{i+1}, but not in a_i. We will form a schedule prime for each element in A as follows.

Let $r_1, r_2 \ldots \in A$. Let R_i be the C-set containing r_i. Let $p_i \in Q_i$. Then we can construct $g_1 \ldots g_i$ for each r_i as follows. Since $p_i \prec p_{i+1}$, we know that f_{i+1} must have a sequentiality index at a_i and Q_{i+1} in R_i's. Let that be R_{n_1}. Then there is a schedule prime g_1 covering f_i such that $g_1 \in a_i : R_{n_1} \stackrel{s}{\rightarrow} Q_{i+1}$ and $g_1 \leq f_{i+1}$. Similarly, there must be a g_2 covering g_1 with the sequentiality index at $a_i \sqcup R_{n_1}$ and the output Q_{i+1} equal to that of f_{i+1} at the same input and the output

C-set. Thus, we can inductively construct the schedule primes between f_i and f_{i+1}.

By Lemma 3.28, there can be no other schedule primes or output primes in f. Hence, the prime approximations to f form a finite chain.

Part II: [$\mathbf{C}_2$] *property.* Since the relation C is defined using the C-set labels, if two elements are related they must have same label; hence the same set of predecessors.

Part III: [**E**] *property.* By definition, we are given two error elements in each C-set as a part of the ppo $\mathcal{P}_{\rightarrow}$; hence the ppo is an error-rich prime basis.

□

The preceding proof establishes that the ppo $\mathcal{P}_{\rightarrow}$ is a prime basis. Now we show that $\mathbf{D}(P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$ is isomorphic to the domain $D_1 \rightarrow_{os} D_2$.

Theorem 3.30 Let $D_1 = \mathbf{D}(P_1, \leq_1, C_1)$ and $D_2 = \mathbf{D}(P_2,\ \leq_2,\ C_2)$. Then, the function space $D_1 \rightarrow_{os} D_2$ isomorphic to the domain $D_{\rightarrow} = \mathbf{D}(P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$. Each set of functions $d \in D_{\rightarrow}$ is identified with its least upper bound in $\mathbf{D}(P_1, \leq_1, C_1) \rightarrow_c \mathbf{D}(P_2,\ \leq_2,\ C_2)$.

Proof.

Part I: *The lub of any prime ideal in $\mathcal{P}_{\rightarrow}$ is an* OS-*function.* Let F be a prime ideal in $\mathcal{P}_{\rightarrow}$. We will show that the least upper bound of F exists and is an *OS*-function.

The least upper bound of the prime ideal exists because all the elements in the prime ideal are consistent with each other. Moreover this least upper bound is a continuous function.

Let $h = \bigsqcup F$. We show that h is a sequential and error-sensitive function, hence an *OS*-function. Let $a \in D_1$ and $h(a)$ be covered by Q^2. We must show that

h has a unique sequentiality index if there is a $b \in D_1$ such that $h(b) \cap Q^2$ is non-empty.

Let $h(b) \cap Q^2 = q$ and $p \prec q$. Then, there must be a prime function $f \in F$ such that $f(a) = p$ and $f(b) = q$, by 3.27. Therefore f must have a sequentiality index R^1 at a and Q^2. Since f is an OS-function, $f(a \sqcup \mathsf{error}_i^{R^1}) = \mathsf{error}_i^{Q^2}$, and hence $\mathsf{error}_i^{Q^2} \in h(a \sqcup \mathsf{error}_i^{R^1})$ for $i = 1, 2$. Therefore R^1 is the sequentiality index at a and Q^2 for the function h. It is also unique, since h is continuous.

Part II: *Any* OS-*function is the lub of a prime ideal in* $\mathcal{P}_{\rightarrow}$. Let the set $S_f = \{p \in P_{\rightarrow} \mid p \sqsubseteq f\}$. Clearly, S_f is downward closed in $P_{\rightarrow}$ and does not contain more than one element from the same C-set. Hence S_f is a prime ideal.

We will show that

$$\forall x \in D_1 : f(x) = \bigsqcup \{p(x) \mid p \in S_f\}.$$

Set $z = \bigsqcup S_f(x)$. It is clear that $z \subseteq f(x)$. To prove the inclusion in the opposite direction, note that by Lemma 3.27, for a prime element $q \in f(x)$, there is a prime function in S_f producing q. Hence, $z \supseteq f(x)$.

□

In this chapter, we described the class of OS-domains using the notion of prime basis. We identified the properties of a domain to be an OS-domain. We defined the class of OS-functions on OS-domains. We showed that the class of OS-functions under the inherited ordering from the space of continuous functions form an OS-domain. Thus, the class of OS-domains along with OS-functions is a possible candidate to model sequential programming languages.

In the next chapter, we will use the OS-domains to build a ccc, that can be used to model λ-calculus based languages.

Chapter 4

Category of OS-domains

In the previous chapter, we described the class of OS-domains. We defined the class of OS-functions on OS-domains and showed that it forms an OS-domain. In this chapter, we will use OS-domains to model programming languages derived from lambda-calculus.

Traditionally, denotational meaning functions are compositional. Semantics of a phrase is given by composing the semantics of its component phrases. The semantics are written as a set of functional equations that map phrases in the programming language into the domains over which the language computes. To justify that a set of semantic functions map the phrases into the appropriate domains, we must provide a proof of validity for the functions. Cartesian closed categories make this process simpler.

We can model any language based on the typed lambda-calculus with cartesian closed categories. If we can prove that the category domains we compute over is cartesian closed, we can use the standard methods to derive a model for higher-order languages. In this section we form a category of OS-domains, and show that it forms a Cartesian Closed Category (ccc).

First, we give some basic definitions for category and cartesian closed category.

Definition 4.1. (*Category*) A *category* $\mathcal{C}$ consists of four collections:

1. a collection of objects, $\mathcal{OBJ}_{\mathcal{C}}$;

2. a collection of arrows, $A \longrightarrow_C B$, for $A, B \in \mathcal{OBJ}_{\mathcal{C}}$;

3. composition operator for arrows, $\circ$, such that $f \circ g$ is an arrow in $A \longrightarrow_C C$ if f is in $A \longrightarrow_C B$ and g is in $B \longrightarrow_C C$;

4. and, for each object A, an object id^A in $A \longrightarrow_C A$ such that id^A is a left and right identity with respect to the composition operator:

$$\mathsf{id}^A \circ f = f \circ \mathsf{id}^A = f.$$

∎

Whenever there is no confusion, we omit the subscripts and superscripts for categories.

Definition 4.2. (*Terminal Object*) An object $\mathsf{1}$ in a category $\mathcal{C}$ is *terminal* iff there exists only one arrow, 1^A in $A \longrightarrow_C \mathsf{1}$. ∎

The following definition introduces a product object.

Definition 4.3. (*Categorical Product*) The object $A_1 \times A_2$ is a *product* of two objects A_1 and A_2 if

- there exists arrows π_i in $A_1 \times A_2 \longrightarrow A_i$ for $i = 1, 2$,

- for any object B and any three arrows f_1 in $B \longrightarrow A_1$ and f_2 in $B \longrightarrow A_2$, and f in $B \longrightarrow A_1 \times A_2$, there exists an arrow $\langle f_1, f_2 \rangle$ in $B \longrightarrow A_1 \times A_2$, the *pair* of f_1 and f_2 such that

$$\begin{aligned} \pi_1 \circ \langle f_1, f_2 \rangle &= f_1 \\ \pi_2 \circ \langle f_1, f_2 \rangle &= f_2 \\ \langle \pi_1 \circ f, \pi_2 \circ f \rangle &= f \end{aligned}$$

  The last equation enforces the uniqueness of the arrow $\langle f_1, f_2 \rangle$.

∎

Definition 4.4. (*Categorical Exponential*) The object B^A is an exponential of two objects A and B if there is an arrow App in $B^A \times A \longrightarrow B$ such that for any object

C and arrow g in $C \times A \longrightarrow B$ there is a unique arrow g^* in $C \longrightarrow B^A$ satisfying the equation:

$$App \circ \langle g^* \circ \pi_1, \pi_2 \rangle = g$$

We call g^* a curried version of g or $\Lambda(g)$. ∎

Definition 4.5. (*Cartesian Closed Category (ccc)*) A category $\mathcal{C}$ is said to be cartesian closed if

1. it has a terminal object,
2. it has a product object for any two objects,
3. it has an exponential object for any two objects.

∎

4.1 Category $\mathcal{OS}$

In this section, we show that OS-domains, along with the OS-functions form a ccc. First we define the category $\mathcal{OS}$.

Definition 4.6. ($\mathcal{OS}$) The category of OS-domains and OS-functions is defined as follows:

1. the collection of objects is the OS-domains;
2. the collection of arrows between D_1 and D_2 is $D_1 \rightarrow_{os} D_2$, the set of OS-functions;
3. the composition operation is the usual function composition as defined in Lemma 3.23; and
4. for each object D the identity function is the identify OS-function in $D \rightarrow_{os} D$.

We use $\mathcal{OS}$ to refer to this category. ∎

Since we have already shown that the composition forms an OS-function and identity is an OS-function, the preceding definition formalizes a category.

4.1.1 $\mathcal{OS}$ is cartesian closed

In this section, we define the product categories and exponentiation. We also show that $\mathcal{OS}$ is cartesian-closed.

The one element domain containing only the $\perp$ is the terminal object. It is easy to see that there is a unique arrow $\mathbf{1}^A : A \longrightarrow 1$ in the category. $\mathbf{1}^A$ is the constant function $\{(x, \perp) \mid x \in A\}$.

Definition 4.7. (*Product* OS-*domain*) Let $D_i = \mathbf{D}(P_i, \leq_i, C_i)$ for $i = 1, 2$. Then we define the domain $D_1 \times D_2$ as $\mathbf{D}(P_\times, \leq_\times, C_\times)$ where

- $$P_\times = \{\langle p_1, \perp\rangle \mid p_1 \in P_1\} \cup \{\langle \perp, p_2\rangle \mid p_2 \in P_2\}$$

  where $\perp$ is a tag representing the empty set;

- $\leq_\times$ is defined as
  $$\begin{aligned} p_1 \leq_1 q_1 &\Rightarrow \langle p_1, \perp\rangle \leq_\times \langle p_2, \perp\rangle \\ p_2 \leq_2 q_2 &\Rightarrow \langle \perp, p_2\rangle \leq_\times \langle \perp, q_2\rangle \end{aligned}$$

- The relation $C_\times$ is defined as
  $$\begin{aligned} (p_1, q_1) \in C_1 &\Rightarrow (\langle p_1, \perp\rangle, \langle q_1, \perp\rangle) \in C_\times \\ (p_2, q_2) \in C_2 &\Rightarrow (\langle \perp, p_2\rangle, \langle \perp, q_2\rangle) \in C_\times \end{aligned}$$

  The C-sets of $C_\times$ are denoted as $\langle Q_i, \perp\rangle$ where Q_i are the C-sets of C_1 and $\langle \perp, R_i\rangle$ where R_i are the C-sets of C_2. In all set operations, we treat $\perp$ as the empty set.

∎

It is easy to verify that $\mathcal{P}_\times$ is a prime basis. Therefore $\mathcal{P}_\times$ generates an OS-domain.

Definition 4.8. (*Projection Functions, Pairing*) For an OS-domain, $D = D_1 \times D_2$, we define the functions $\pi_i \in D_1 \times D_2 \rightarrow_{os} D_i$ as follows:

$$\pi_1(x) = \{p \mid \langle p, \perp\rangle \in x\}$$

$$\pi_2(x) = \{q \mid \langle \perp, q\rangle \in x\}$$

It is easy to verify that π_i are OS-functions.

An element c of the domain is written as $\langle a, b\rangle$ where $a = \pi_1(c)$ and $b = \pi_2(c)$. It is easy to verify that a is in D_1 and b is in D_2.

Also, for $f_i \in D \rightarrow_{os} D_i$ for $i = 1, 2$, define the function $\langle f_1, f_2\rangle \in D \rightarrow_{os} D$ as

$$\langle f_1, f_2\rangle(x) = \langle f_1(x), f_2(x)\rangle$$

It is easy to see that the pair is an OS-function. ∎

Lemma 4.9 For any two objects A and B in the category $\mathcal{OS}$, there is a product object.

Proof. We define the product object of A and B as $A \times B$ and show that there are projections and pairing of functions.

The projection functions are π_1 and π_2. We must show the following conditions.

- $\pi_1 \circ \langle f_1, f_2\rangle = f_1$ **and** $\pi_2 \circ \langle f_1, f_2\rangle = f_2$

  $$(\pi_1 \circ \langle f_1, f_2\rangle)(x) = \pi_1(\langle f_1(x), f_2(x)\rangle) = f_1(x)$$

  By extensionality, we have the desired result:

  $$\pi_1 \circ \langle f_1, f_2\rangle = f_1$$

  Similarly,

  $$\pi_2 \circ \langle f_1, f_2\rangle = f_2$$

- $\langle \pi_1 \circ f, \pi_2 \circ f\rangle = f$ **for** $f \in C \longrightarrow A \times B$. This shows the uniqueness of the arrow $\langle f_1, f_2\rangle$. Since

  $$\langle \pi_1 \circ f, \pi_2 \circ f\rangle(c) = \langle \pi_1(f(c)), \pi_2(f(c))\rangle$$

  and if $f(c) = \langle a, b\rangle$, by the definitions of π_i, we have

  $$\langle \pi_1 \circ f, \pi_2 \circ f\rangle(c) = \langle a, b\rangle = f(c).$$

Therefore, by extensionality,

$$\langle \pi_1 \circ f, \pi_2 \circ f \rangle = f$$

Therefore $\mathcal{OS}$ has categorical products. $\square$

Definition 4.10. (*Exponential, Apply*) For two objects A and B, we define the B^A as the domain $A \rightarrow_{os} B$, which is an OS-domain. We define the function $Apply_{AB}$ in $B^A \times A \rightarrow B$ as

$$Apply_{AB}\langle f, a \rangle \stackrel{df}{=} f(a)$$

∎

Whenever there is no confusion, we drop the subscripts on *Apply*.

Lemma 4.11 $Apply_{AB}$ is an OS-function.

Proof.

Apply **is continuous.** The following argument shows that *Apply* is continuous in both the arguments.

Let	a	$\sqsubseteq$	$\bigsqcup a_i$	$a_i \in A$.
Then	$f(a)$	$\sqsubseteq$	$\bigsqcup f(a_i)$	since$f$ is continuous.
Therefore	$apply\langle f, a \rangle$	$\sqsubseteq$	$\bigsqcup apply\langle f, a_i \rangle$.	
Similarly let	f	$\sqsubseteq$	$\bigsqcup f_i$	$f_i \in B^A$.
Since	$f(a)$	$\sqsubseteq$	$\bigsqcup f_i(a)$	
we have	$apply\langle f, a \rangle$	$\sqsubseteq$	$\bigsqcup apply\langle f_i, a \rangle$.	

Therefore, *Apply* is continuous.

Apply **is sequential.** Let $apply\langle f, a \rangle \prec Q$. Also, let $apply\langle f', a' \rangle \cap Q$ be non-empty and $\langle f', a' \rangle \sqsupseteq \langle f, a \rangle$. We will show that there is a sequentiality index $si_{apply}(\langle f, a \rangle, Q)$.

By the definition of *apply*, is is easy to see that $f(a) \prec Q$ and $f'(a') \cap Q \neq \emptyset$. We analyze the function f as follows.

Case 1. $\mathsf{error}_i^Q \in f(a \sqcup \mathsf{error}_i^R)$ **for some** R **covering** a**:** If such an R exits, it must be unique, and it the sequentiality index of f at (a, Q). By Lemma 3.27, we know that there must be a prime $g \in f \cap (a : R \stackrel{s}{\rightarrow} Q)$.

We will show that the sequentiality index of *apply* at $(\langle f, a\rangle, Q)$ is $\langle \perp, R\rangle$. Let $f_2(a_2) \cap Q$ be q, for some $\langle f_2, a_2\rangle \sqsupseteq \langle f, a\rangle$. If $a_2 \cap R = \emptyset$, then $f_2(a_2 \sqcup \mathsf{error}_i^R) \sqsupseteq q \sqcup \mathsf{error}_i^Q$ which is not possible: q cannot be consistent with *both* error elements of Q, since $q \in Q$. Therefore, $a_2 \cap R$ is non-empty. Hence, $\langle f_2, a_2\rangle \cap \langle \perp, R\rangle \neq \emptyset$. [11]

Case 2: $\mathsf{error}_i^Q \notin f(a \sqcup \mathsf{error}_i^R)$ **for any** R **covering** a**:** Let $p \prec Q$. Let $a_h \sqsubseteq a$ be the threshold for p in f. There must be an output prime $h \in f \cap a_h \stackrel{o}{\rightarrow} p$. Obviously h is the maximal output prime in f with output in the direction of Q, by the premise of this case.

Let G be the set of schedule primes in f above h belonging to $a_i : R_i \stackrel{s}{\rightarrow} Q$, for $i = 1, 2, \ldots$. Here R_i are the C-sets in the input; they are the direction in which the input can be further explored.

Let h' be a prime in $a \stackrel{o}{\rightarrow} q$ where $q \in Q$ such that $h' \uparrow f$. It is easy to verify that the set G forms a chain under h', and a_i are strictly below a.

Case a. G **is non-empty:** Let g be the schedule prime such that a_g is maximal in the set $\{a_i\}$. Let $g \in a_g : R_g \stackrel{s}{\rightarrow} Q$. We know that $a_g \sqsubseteq a$. Since $g \in f$, and by the premise of this case, $R_g \cap a \neq \emptyset$. If the premise is not true, then g could be a schedule prime in $a : R \stackrel{s}{\rightarrow} Q$.

Let $a_* = a_g \sqcup (R_g \cap a)$ and $F' = g$. Let R_* be the C-set with the label $\langle a_*, Q, F'\rangle$.

Case b. G **is empty:** In this case let R_* be the C-set with the label $\langle a_*, Q, F'\rangle$ where $a_* = a_h$, $F' = h$.

[11] As stated earlier, $\perp$ is treated as an empty set in the set operations.

In either case R_* is the C-set label of the maximal prime, (schedule or output) that approximates h' and is contained in in f. Now, we show that the sequentiality index of *Apply* at $\langle f, a\rangle$ and Q is $\langle R_*, \perp\rangle$.

Let q be $Apply\langle f_1, a_1\rangle \cap Q$, for some $\langle f_1, a_1\rangle \sqsupseteq \langle f, a\rangle$. We must show that $f_1 \cap R_*$ is non-empty. Let a_2 be the threshold for q in f_1. Then, by Lemma 3.27 there must be a prime $g' \in a_2 \xrightarrow{o} q \cap f_1$. And, by Lemma 3.28, $g' \sqsupseteq F'$. We will show that there is a prime function in R_* approximating g'.

- $a_2 = a_*$: By Lemma 3.28, g' belongs to R_*, concluding the proof.
- $a_2 \sqsupset a_*$: Since g' is an OS-function it must have a sequentiality index, $R_{g'}$, at (a_*, Q). Therefore g' must contain a schedule prime in a_* : $R_{g'} \xrightarrow{s} Q$, which is in R_*.

Therefore, $\langle R_*, \perp\rangle$ is the sequentiality index of *Apply* at $(\langle f, a\rangle, Q)$.

Apply **is error-sensitive:** We use the preceding case analysis (1 and 2) to show that in either case, *Apply* is error-sensitive.

Case 1: The sequentiality index is $\perp \times R$: Since f has the schedule prime with $si(a, Q) = R$, $f(a \sqcup \mathsf{error}_i^R)$ must have error_i^Q. Therefore, we have the required result, i.e, $\mathsf{error}_i^Q \in apply(\langle f, a\rangle \sqcup \langle \perp, \mathsf{error}_i^R\rangle)$.

Case 2: The sequentiality index is $R_* \times \perp$: The error elements F_{e_1}, F_{e_2} in R_* are in $a_* \xrightarrow{o} \mathsf{error}_1^Q$ and $a_* \xrightarrow{o} \mathsf{error}_2^Q$ respectively. Therefore, $\mathsf{error}_i^Q \in (f \sqcup F_{e_i})(a)$. Hence, we have $\mathsf{error}_i^Q \in apply(\langle f, a\rangle \sqcup \langle F_{e_i}, \perp\rangle)$.

Thus, we conclude that *apply* is an OS-function, hence an arrow in the category $\mathcal{OS}$. $\square$

Theorem 4.12 The category $\mathcal{OS}$ is cartesian closed.

Proof. $\mathcal{OS}$ has a terminal object, the domain containing only $\perp$. Also, by Lemma 4.9, $\mathcal{OS}$ has categorical products.

For any two objects A, B, the object B^A is an exponential object. Let $Apply$ be the function defined in the definition 4.10. We will show that for any g in $C \times A \rightarrow_{os} B$, there is a unique arrow g^* in $C \rightarrow_{os} B^A$ such that

$$Apply \circ \langle g^* \circ \pi_1, \pi_2 \rangle = g.$$

For ease of notation, consider $g(c, a)$ to be $g(\langle c, a \rangle)$. Define $g^*(c)(a) = g(c, a)$. That is $g^*(c)$ is a function that takes an element a to $g(c, a)$. We have to prove that g^* is an arrow in the category $\mathcal{OS}$.

- g^* **is a member of** $C \rightarrow_{os} B^A$**.** This proof has two parts. First, we must show that $g^*(c)$ is a member of B^A for any $c \in C$. Let $g^*(c) = f$. We will show that f is in $A \rightarrow_{os} B$.

  f **is continuous.**

$$\begin{array}{llcl} \forall a \in A & f(a) & = & g(c, a) \\ \text{for a chain } a_i & f(\bigsqcup a_i) & = & g(c, \bigsqcup a_i) \\ & & = & \bigsqcup g(c, a_i) \\ & & = & \bigsqcup f(a_i) \end{array}$$

  Therefore f is continuous.

  f **is sequential.** To check for sequentiality, define $si_f(a, Q) = \pi_2(si_g(\langle c, a \rangle, Q))$. If Q covers $f(a)$ then it covers $g(\langle c, a \rangle)$ also. And, if there is an $a' \sqsupseteq a$ such that $f(a') \cap Q \neq \emptyset$ then it follows that $\langle c, a' \rangle \cap \langle \perp, Q \rangle \neq \emptyset$ Hence, if $si_g(\langle c, a \rangle, Q)$ covers $\langle c, a \rangle$, $si_f(a, Q)$ covers a. Therefore, the sequentiality index for f is properly defined.

  If $f(x) \cap Q = q$ and $x \sqsupseteq a$, then $g(\langle c, x \rangle) \cap Q = \langle \perp, q \rangle$. Since g is an OS-function, we have $\langle c, x \rangle \cap \langle \perp, R \rangle \neq \emptyset$ where $\langle \perp, R \rangle = si_g(\langle c, a \rangle, Q)$. Therefore by the preceding definition of si_f, we have the desired proof of sequentiality of f, that is $x \cap R \neq \emptyset$.

f is error-sensitive. We can verify the error-sensitivity as follows. Let $R = si_f(a, Q)$. Then $f(a \sqcup \mathsf{error}_i^R) = g(c \times a \sqcup \mathsf{error}_i^R) = \mathsf{error}_i^Q$, by the error-sensitivity of g.

Now,we will show that g^* is an OS-function in $C \rightarrow_{os} B^A$.

g^* is continuous.

$$\begin{array}{lrcl} \forall a \in A & g^*(c) & = & g(c, a) \\ \text{for a chain } c_i & g^*(\bigsqcup c_i) & = & g(\bigsqcup c_i, a) \\ & & = & \bigsqcup g(c_i, a) \\ & & = & \bigsqcup g^*(c_i) \end{array}$$

Therefore g^* is continuous.

g^* is sequential. Let $g^*(c)$ be covered by Q^*, and assume that for some $c' \sqsupseteq c$, $g^*(c') \cap Q^*$ is non-empty. Let the C-set label of Q^* be $\langle a, Q, F' \rangle$. Then we know that $g^*(c)(a)$ is covered by Q, i.e., $g(c \times a)$ is covered by Q. Similarly, since $g^*(c')(a) \cap Q \neq \emptyset$, we know that $g(c' \times a) \cap Q$ is non-empty for some $c' \sqsupseteq c$. Since g is an OS-function, there must be a sequentiality index $R_c \times \perp = si_g(c \times a, Q)$. We define $si_{g^*}(c, Q^*) = R_c$.

Since R_c covers c and Q^* covers $g^*(c)$, the sequentiality index is well-defined. And, if there is a $x \sqsupseteq c$ and $g^*(x) \cap Q^* \neq \emptyset$, then $g^*(x)(a) \cap Q \neq \emptyset$. Since $g(x \times a) \cap Q \neq \emptyset$, we know that $x \times a \cap R_c \times \perp$ is non-empty; hence $x \cap R_c$ is non-empty, proving that R_c is the sequentiality index for g^* at (c, Q^*).

g^* is error-sensitive. The error sensitivity can easily be verified as follows. Let $R_c = si_{g^*}(a, Q^*)$. Then, we know that $R_c \times \perp = si_g(a \times c, Q)$. Since g is error-sensitive, $\mathsf{error}_i^Q \in g((c \sqcup \mathsf{error}_i^{R_c}) \times a)$. Therefore $\mathsf{error}_i^Q \in g^*(c \sqcup \mathsf{error}_i^{R_c})(a)$. That means, $g^*(c \sqcup \mathsf{error}_i^{R_c}) \cap a \overset{o}{\rightarrow} \mathsf{error}_i^Q = F_i$ for $i = 1, 2$. These two functions are error primes and they cover F' by Lemma 3.28.

- g^* **is a curried version of** g. Since

$$apply \circ \langle g^* \circ \pi_1, \pi_2 \rangle (\langle c, a \rangle) = apply(g^*(c), a)$$

and,

$$g^*(c)(a) = g(c, a),$$

we have the required result.

- g^* **is unique.** To show the uniqueness of g^*, let g' satisfy

$$apply \circ \langle g' \circ \pi_1, \pi_2 \rangle = g'$$

That is,

$$apply \circ \langle g' \circ \pi_1, \pi_2 \rangle (\langle c, a \rangle) = apply(g'(c), a).$$

Since, by the premise,

$$g'(c)(a) = g(c, a).$$

Since,

$$g^*(c)(a) = g(c, a),$$

by extensionality,

$$g' = g^*$$

thus proving the uniqueness of g^*.

□

In addition to the usual categorical functors, we define the following functors, $+$ and *lift*.

Definition 4.13. (*Sum*) Given two *OS*-domains $\mathbf{D}(P_1, \leq_1, C_1)$ and $\mathbf{D}(P_2, \leq_2, C_2)$ their sum is defined as follows.

The prime elements:

$$P_+ = \{\mathsf{error}_1, \mathsf{error}_2\} \cup \{\langle x, 1 \rangle \mid x \in P_1 \cup \{\bot\}\} \cup \{\langle y, 2 \rangle \mid y \in P_2 \cup \{\bot\}\}.$$

That is, two $\bot$'s are added to the prime elements.

The ordering relation: It is the minimal partial order satisfying the following:

$$\begin{array}{lll} \langle x,1\rangle & \leq_+ & \langle x',1\rangle \quad \text{if } x \leq_1 x' \text{ in } P_1 \text{ or } x=\bot \\ \langle y,2\rangle & \leq_+ & \langle y',2\rangle \quad \text{if } y \leq_2 y' \text{ in } P_2 \text{ or } y=\bot \end{array}$$

The conflict relation:

$$\begin{array}{lll} (\langle x,1\rangle,\langle x',1\rangle) \in C_+ & \iff & (x,x') \in C_1 \\ (\langle y,2\rangle,\langle y',2\rangle) \in C_+ & \iff & (y,y') \in C_2 \end{array}$$

In addition, $\{\mathsf{error}_1, \mathsf{error}_2, \langle\bot,1\rangle, \langle\bot,2\rangle\}$ is a C-set.

It is easy to see that the preceding ppo is an error-rich prime basis. $A+B$ is an OS-domain with $\mathsf{error}_1, \mathsf{error}_2$ being the errors above $\bot$ and the following order: To turn $+$ into a functor, we define it on the arrows also. Given $f : A \rightarrow_{os} A'$ and $g : B \rightarrow_{os} B'$, define $f \oplus g : (A+B) \rightarrow_{os} (A'+B')$ by:

$$\begin{array}{rcl} (f\oplus g)\bot & = & \bot \\ (f\oplus g)\mathsf{error}_i & = & \mathsf{error}_i \\ (f\oplus g)\langle x,1\rangle & = & \langle f(x),1\rangle \\ (f\oplus g)\langle y,2\rangle & = & \langle g(y),2\rangle \end{array}$$

With these definitions $+$ is a covariant functor on the category of OS-domains. Even though the usual injection is an arrow in this space, the projection is not. ∎

Definition 4.14. (*Lift*) Given an OS-domain $\mathbf{D}(P,\leq,C)$ define its *lift*, an OS-domain with prime basis $(P_\odot, \leq_\odot, C_\odot)$ as follows.

The prime elements:

$$P_\odot = \{\mathsf{error}_1, \mathsf{error}_2, \langle x,0\rangle \mid x \in P \cup \{\bot\}\}$$

The ordering relation: It is the minimal partial order satisfying the following:

$$\langle x,0\rangle \leq_\odot \langle x',0\rangle \quad \text{if } x \leq x' \text{ in } P \text{ or } x=\bot$$

The conflict relation: It is the minimal equivalence relation satisfying the following:

$$
\begin{aligned}
(\langle x,0\rangle, \langle x',0\rangle) \in C_{\odot} &\iff (x,x') \in C \\
(\mathsf{error}_1, \mathsf{error}_2) \in C_{\odot} & \\
(\langle \bot, 0\rangle, \mathsf{error}_1) \in C_{\odot} &
\end{aligned}
$$

It is easy to verify that the preceding ppo is an error-rich prime basis.

To make the lift operator a functor, define it on any $f : A \rightarrow_{os} A'$,

$$
\begin{aligned}
\odot(f)(\bot) &= \bot \\
\odot(f)(\mathsf{error}_i) &= \mathsf{error}_i \\
\odot(f)\langle x, 0\rangle &= \langle f(x), 0\rangle \quad \forall x \in A
\end{aligned}
$$

Similar to the + functor, this $\odot$ does not have the *force* arrow that brings the original domain back. Such a function will have to force the additional errors in the lifted domain to the bottom element and hence cannot be error-sensitive. ∎

The categorical operators are useful in defining a broad class of domains. In practice, these operators are sufficient to construct all the necessary domains used in PCF-like language. In addition *lift* and + are used to provide semantics of lazy evaluation [CD82]. However, these operators define only a limited class of domains. In the next chapter, we introduce a more general method that is based on the universal domain.

4.2 Categorical Model

We show how the *OS*-domains can be used in providing semantics for languages like PCF. It is a language based on typed lambda-calculus. It includes primitive types such as integers and induction scheme to build the functional types. It also contains primitive operations to operate on the base types.

Historically, PCF served an important role in the study of programming languages. It is used to illustrate concepts of denotational and operational semantics.

Traditionally the semantic model for PCF is provided using the continuous function space for function types. In this section we show a model using the OS-function space for that purpose.

The cartesian closed category $\mathcal{OS}$ provides a denotational model for PCF [Bar84, Koy84]. In the following sections we introduce the language PCF and provide the semantics for the language. Unlike the original PCF, we use PCF with only one ground type, the integers. We use test for 0 for the conditional statement in the *if* construct.

The first part of Figure 4.1 defines the set of syntactically well-formed PCF phrases. A PCF phrase M is either a constant, *i.e.*, a numeral $\ulcorner n \urcorner$, $n \geq 0$, a functional constant, a typed variable x^τ, a λ-abstraction, or an application.

The second part of the Figure 4.1 presents the possible types. They consist of a ground type o and infinite number of finitely generated types of the form $\sigma \to \tau$.

The typing rules are presented in the last part of the 4.1. These rules specify a unique type for a valid PCF phrase.

Finally, programs are closed expressions of ground type.

4.2.1 Denotational Semantics

In this section we define a model of PCF using OS-domains. The semantics assigns meanings to types and phrases by mapping them to appropriate mathematical entities. By making sure that these mathematical entities satisfy certain properties, we can provide a model where the semantics assigns proper meaning to the fix point operator Y. The category $\mathcal{OS}$ satisfies these properties.

Now we define the model in the following way. Define D^i to be $\mathbf{N}_e$. Let $D^{\sigma\to\tau}$ be $[D^\sigma \to_{os} D^\tau]$. Let Env be the infinite product of D^σ's indexed by x^σ. Let $E^\sigma = [\text{Env} \to_{os} D^\sigma]$. Now define the meaning function over the constants in the following

Syntax:

$$
\begin{array}{rcl}
M & ::= & c \mid x \mid (\lambda x.M) \mid (M\ M) \\
c & ::= & \ulcorner n \urcorner \mid \mathbf{1}^{+} \mid \mathsf{sub1} \mid \mathsf{if0} \mid \mathsf{Y}^{\tau} \\
x & ::= & x^{\tau} \mid y^{\tau} \mid \ldots
\end{array}
$$

Types: $\sigma, \tau ::= o \mid (\tau \rightarrow \sigma)$

Type Checking:

$$
\frac{A, x^{\tau} \vdash M : \tau'}{A \vdash \lambda x^{\tau}.M : \tau \rightarrow \tau'} \qquad \frac{A \vdash M : \tau' \rightarrow \tau; \quad A \vdash M' : \tau'}{A \vdash (MM') : \tau}
$$

$$
\begin{array}{lll}
A \vdash x^{\tau} : \tau \text{ if } x^{\tau} \in A & A \vdash \ulcorner n \urcorner : o \text{ for all } n & A \vdash \mathsf{Y}^{\tau} : (\tau \rightarrow \tau) \rightarrow \tau \\
A \vdash \mathbf{1}^{+} : o \rightarrow o & A \vdash \mathsf{sub1} : o \rightarrow o & A \vdash \mathsf{if0} : o \rightarrow o \rightarrow o \rightarrow o
\end{array}
$$

Figure 4.1 PCF: Syntax

way:

$$
\begin{array}{rcl}
\mathbf{E}[\![n]\!] & = & \ulcorner n \urcorner \\
\mathbf{E}[\![add1]\!](x) & = & \begin{cases} \bot & \text{if } x = \bot \\ \mathsf{error}_i & \text{if } x = \mathsf{error}_i \\ \ulcorner x+1 \urcorner & \text{otherwise} \end{cases} \\
\mathbf{E}[\![sub1]\!](x) & = & \begin{cases} \bot & \text{if } x = \bot \\ \mathsf{error}_i & \text{if } x = \mathsf{error}_i \\ \ulcorner 0 \urcorner & \text{if } x = 0 \\ \ulcorner x-1 \urcorner & \text{otherwise} \end{cases} \\
\mathbf{E}[\![if0]\!](p)(x)(y) & = & \begin{cases} \bot & \text{if } p = \bot \\ \mathsf{error}_i & \text{if } p = \mathsf{error}_i \\ x & \text{if } p = 0 \\ y & \text{otherwise} \end{cases} \\
E[\![\mathsf{Y}^{\tau}]\!](f) & = & \bigsqcup_{n \geq 0} f^{n}(\bot)
\end{array}
$$

Using this interpretation, we will provide a meaning function to give denotations to the phrases in the language. For the we define two auxiliary functions:

$$\pi_{x^\sigma} : \mathrm{Env} \rightarrow_{os} D^\sigma$$

such that it projects the value corresponding to x^σ,

$$S_{x^\sigma} : \mathrm{Env} \times D^\sigma \rightarrow_{os} \mathrm{Env}$$

such that $\pi_x \circ S_x = \pi_2$ and $\pi_y \circ S_x = \pi_y \circ \pi_1$. These two functions are observably sequential, hence they are functions in the domains.

$$\begin{aligned}
[\![x]\!] &= \pi_x \\
[\![c]\!] &= \bot_{\mathrm{Env}} \rightarrow_{os} \mathbf{E}[\![c]\!] \\
[\![(MN)]\!] &= apply \circ \langle [\![M]\!], [\![N]\!] \rangle \\
[\![(\lambda x.M)]\!] &= ([\![M]\!] \circ S_x)^{*12}
\end{aligned}$$

All the above equations are valid because the domain is cartesian closed. Moreover, all the primitive functions as OS-functions.

The model provided by OS-domains satisfies many properties. In particular, it is an order-extensional, least fix point model. We define these notions as follows.

Definition 4.15. (η-*model*) A model is anη-model if it satisfies

$$(\eta) \quad [\![\lambda x.Mx]\!] \;=\; [\![M]\!] \quad \text{if } x \text{ is not free in } M$$

∎

Definition 4.16. (*Extensional Model*) A model is extensional if the following properties hold.

$$\begin{array}{lrl}
\text{(environment extensional)} & \forall \rho : e\rho = e'\rho & \Rightarrow e = e' \\
\text{(value extensional)} & \forall d'' : apply(d, d'') = apply(d', d'') & \Rightarrow d = d'
\end{array}$$

∎

The classical model defined in [Sto81] is an extensional model. The stable function models as in [Ber79] based on the stable ordering is also extensional. However, the sequential algorithms model is not extensional. By adding the error elements, it can be made order-extensional [Cur92]. Cartwright et. al [CCF94] consider various categories that can be obtained by adding 0, 1, and 2 errors to the domain. Without errors, the model is not extensional. By adding a single error, the model can be made into an extensional one. With two errors, the model becomes order-extensional.

Definition 4.17. (*Order-extensional Model*) A model is order-extensional if the following properties hold.

$$\begin{array}{lrl} \text{(environment order-extensional)} & \forall \rho : e\rho \leq e'\rho & \Rightarrow e \leq e' \\ \text{(value order-extensional)} & \forall d'' : apply(d, d'') \leq apply(d', d'') & \Rightarrow d \leq d' \end{array}$$

∎

Again, the classical model is order-extensional, but the stable model and the sequential algorithms are not order-extensional.

Definition 4.18. (*Least Fixpoint Model*) A *least fixpoint model* is a continuous model such that the $[\![Y]\!]$ is the least fixpoint operator, *i.e.*, $[\![Y]\!]\bot d$ is a least fixpoint of the function defined by d. ∎

In the classical model of PCF [Sto81], D^i is $\mathbf{N}_\bot$. For any type $\sigma \rightarrow \tau$, $D^{\sigma\rightarrow\tau}$ is the set of continuous functions from D^σ to D^τ. For any type σ, E^σ is ENV $\rightarrow D^\sigma$. The constants map to the obvious choices and Y is interpreted as the least fixpoint operator that is guaranteed to exist. The equations var, app, lam define the meaning of the terms. This model is a least fix point, order extensional η-model.

The stable functions build the model similar way using the stable ordering over the stable functions instead of pointwise ordering over the continuous functions. This model is an extensional model, but not order-extensional. In the case of sequential algorithms the construction is slightly more complicated. The domain D^i is a concrete data structure [BC82] of integers. The domain $D^{\sigma\rightarrow\tau}$ is the set of sequential algorithms from D^σ to D^τ. The constants are assigned suitable algorithms as the meaning. Y defines a least fixpoint algorithm in this model. Under this scheme, any term of type σ is assigned the meaning in ENV $\rightarrow_{sa} D^\sigma$, i.e., an algorithm in that domain. This model is neither environment-extensional nor value-extensional. It satisfies the η rule, hence it is an η-model.

Theorem 4.19 The OS-domain model satisfies the following properties:

(β) $[\![(\lambda x.M)N]\!] = [\![M[x/N]]\!]$.

(Least fixpoint) Y^τ is the least fix point operator.

(Order-extensional) The model is environment and value order-extensional.

Proof. The properties (β),**(Least fixpoint)** are proved as in [Cur93]. These properties do not rely on the underlying domains. Order-extensionality depends on the extensional behavior of OS-functions. Environment order-extensionality is trivial. Consider value order-extensionality:

$$\forall d'' : apply(d, d'') \leq apply(d', d'') \Rightarrow d \leq d'$$

This property holds because the OS-function space is ordered under the usual pointwise ordering. □

Thus, we can provide an order-extensional model for PCF using OS-domains. However, this model is not fully abstract. In particular, this model assigns different meanings based on the function evaluation order, whereas the language cannot differentiate the order of evaluation in a function. In Chap. 7, we will present a modification of PCF to include such capabilities.

Chapter 5

Universal OS-domains

In previous chapters, we defined the category of OS-domains and OS-functions. Since this category is cartesian-closed, we can use it to construct models for languages based on the typed lambda-calculus. This chapter focuses on the more ambitious problem of devising a framework for effectively defining arbitrary computable OS-domains and OS-functions. In such a framework, we can define any effective model based on OS-domains and OS-functions.

As we mentioned in Chapter 2, such a framework consists of a universal domain U and a corresponding universal language L_U where every computable element of U is *effectively* defined by a program in L_U. To a mathematician familiar with category theory, it might appear that we can rely on the "universal object" construction for ccc's to build this framework. Every ccc satisfying certain constraints [Sco80, BC82, Dro89, Dro91] contains a universal reflexive object in which every object can be "embedded". The construction closely follows the inverse limit construction for solving domain equations. However, this approach does not suffice for two reasons. First, the construction ignores the issue of preserving the essential computational properties (*e.g.* the finite maximality of elements) of embedded domains. Second, the construction does not yield any insight on how to design a universal language for effectively defining all computable elements of the domain.

In the literature on domain theory, the precise definition of "universality" of domains varies along three "axes":

- the class of domains,
- the class of functions used to embed domains, and

- the properties preserved by the embeddings.

In our case, the class of domains is the OS-domainsand the class of embedding functions is the OS-functions. Domain embeddings must preserve the ordering, consistency, finiteness, and finite maximality of embedded elements. The finiteness and finite maximality properties are included so that computations over an arbitrary domain can be simulated by computations over the universal domain. Therefore, it is clear that a solution based on ccc is not sufficient for our purposes.

The chapter is divided into four sections. The first section reviews previous formulations of the universal domain and identifies their limitations. The second section constructs a new universal domain $\mathbf{T}$ satisfying all of the properties identified above except one: preserving the finiteness of embedded elements. In the third section, we prove that $\mathbf{T}$ has the claimed properties. Finally, in the last section, we refine the structure of $\mathbf{T}$ so that the finiteness of embedded elements is preserved.

The preliminary results of this chapter have been presented in [KC93].

5.1 Universal Domains – An Introduction

The core of Domain Theory consists of two constructions:

- a universal domain U of data objects in which any data domain can be embedded as subdomain,
- a universal language L_U that can express any computable element of D.

A universal domain U uses functions over U to model higher-order computation. For example, classical universal domains like T^ω model higher order computation using *continuous functions*. The corresponding universal language L_U can express all the *computable continuous* functions.

Using the framework of a universal domain, we can define the meaning of a programming language by embedding the language syntax S and the data domain D in

U. The meaning of a programming language is a function that maps phrases in the programming language into the data domain D.

The embedding of a data domain D in U is achieved by identifying an isomorphic subdomain D' within the universal domain. A domain $(D, \sqsubseteq_D)$ is a subdomain of a domain $(U, \sqsubseteq_U)$ iff

- $D \subseteq U$,
- $\sqsubseteq_D = \sqsubseteq_U \cap (D \times D)$.

Alternately, a subdomain D of the universal domain U can be described by a *retraction* over U, a function that coerces all of elements of U to elements in the embedded image of D.

Definition 5.1. (*Retraction, Retract*) A continuous function f is a *retraction* of U iff $f \circ f = f$, where $\circ$ is the composition operator. The corresponding cpo $Dom(f) = (\{x \mid f(x) = x\}, \sqsubseteq)$, where $\sqsubseteq$ is inherited from D, is called a *retract* of U. ∎

A retract is not necessarily a domain because it may not satisfy the property of ω-algebraicity. Hence, it is customary to deal with retractions that are partial closures [Plo78].

Definition 5.2. (*Partial Closure*) A retraction f is a partial closure iff $Dom(f)$ is ω-algebraic. ∎

It can easily be shown that projections, retractions that approximate the identity function, are partial closures. In addition, any retraction that maps finite elements to finite elements is a partial closure.

Given a data domain D, we can embed it in U by finding a retraction f_D. However, it raises a question: how do we *present* a domain? What is the mathematical representation of a domain?

When we denote the finite elements of a domain using canonical names, each name can be interpreted as the index of the corresponding element. Since these names are constructed as finite strings over a finite alphabet, they can be placed in one-to-one correspondence with a subset of the natural numbers. Given the indices of elements, it is easy to compute relations such as $\sqsubseteq$, $\uparrow$, and $\sqcup$. In the language of recursion theory, these elementary relations on indices are recursive.

We can formalize this idea using the concept of a domain presentation [Sco81]:

Definition 5.3. (*Scott Presentation*) A *presentation* δ_s of a Scott domain D with basis $(B, \leq)$ is a triple $\langle \mathbf{E}_s, Lub, Con \rangle$ consisting of:

- a one-to-one onto function $\mathbf{E}_s : B \to \mathcal{I} \subseteq \mathbf{N}$, such that $\mathbf{E}_s(\perp) = 0$; $\mathbf{E}_s$ maps the elements of B into their codes in the index set $\mathcal{I} \subseteq \mathbf{N}$.

- a ternary predicate $Lub \subseteq \mathcal{I} \times \mathcal{I} \times \mathcal{I}$ such that

$$Lub(x, y, z) \iff b_x \sqcup b_y = b_z$$

  where $b_x = E^{-1}(x), b_y = E^{-1}(y)$, and $b_z = E^{-1}(z)$;

- a binary predicate $Con \subseteq \mathcal{I} \times \mathcal{I}$ such that

$$Con(x, y) \iff \exists z \in B : \; x \sqsubseteq z \text{ and } y \sqsubseteq z \,.$$

The function $\mathbf{E}_s$ is called the *enumerator* for D. The presentation δ_s is *effective* iff the relations Lub and Con are partial recursive. ∎

The enumerating function $\mathbf{E}_s$ assigns a natural number to each finite element. The predicates on these natural numbers encode the information about the structure of domain.

For any domain D given by a presentation δ_s, we can identify an isomorphic subdomain D' in the universal domain U by defining an appropriate *retraction* f_D on U. Moreover, if the presentation is effective (the three relations $\mathbf{E}_s, Lub, Con$ are recursive, we can effectively construct f_D.

Using such a formulation of domain theory, we can write denotational definitions for programming languages in the universal language L_U. More precisely, we can define a meaning function in L_U that maps program phrases, embedded as elements of U, to their meanings, embedded as elements of U. This meaning function looks like an abstract interpreter written in L_U.

Unfortunately, in classical domain theory, the meaning function is not an effective interpreter because domain embeddings in U do not preserve enough computational information. In particular, they fail to stipulate when computations terminate. The finite maximal elements of an embedded domain D—which correspond to terminating computations over D—are not mapped to finite maximal elements in U. In fact, the classical domains T^ω and $\mathsf{P}\omega$ do not contain any finite maximal elements. As a result, all computations expressed in the corresponding universal languages must not terminate. This issue is discussed in detail in [CD88].

5.1.1 ∇ – A Universal Domain for Computation

For a universal domain to support arbitrary higher order computation, the corresponding domain embeddings must preseve the finite maximality of elements. In an attempt to address this issue, Cartwright and Demers constructed a new universal domain called ∇ [CD88]. In ∇, domain embeddings are projections that preserve finiteness and maximality of elements. The easiest way to define ∇ is to construct the complete lattice $\nabla^\top$ and remove the maximum element *top* to form the cpo ∇. $\nabla^\top$ is a domain of trees over $\bot, \odot, \top$ such that the trees are collapsed under the rules:

$$\begin{array}{rcl}
\langle \top, \top \rangle & \rightarrow & \top \\
\langle \bot, \bot \rangle & \rightarrow & \bot \\
\langle \odot, \bot \rangle & \rightarrow & \bot \\
\langle \bot, \odot \rangle & \rightarrow & \bot \\
\langle \odot, \odot \rangle & \rightarrow & \bot
\end{array}$$

The order relation is defined on the collapsed trees, *i.e.*, the canonical trees such that:

$$\begin{aligned} \odot &\sqsubseteq \odot \\ \odot &\sqsubseteq \top \\ \top &\sqsubseteq \top \\ \forall\, x : \bot &\sqsubseteq x \\ \forall a, b, a', b' : \langle a, b\rangle &\sqsubseteq \langle a', b'\rangle \iff a \sqsubseteq a' \wedge b \sqsubseteq b'. \end{aligned}$$

∇, which is $\nabla^{\top} - \{\top\}$, is a domain with finite maximal elements. If a domain D is presented with maximality information, *i.e.*, the presentation is like scott presentation with an additional predicate max

$$\max(i) \iff \not\exists k : \alpha_i \sqsubseteq \alpha_k,$$

then we can produce a continuous function f_D over ∇ with the following properties:

- f_D is a projection, *i.e.*, for all $x \in U$ $f_D(x) \sqsubseteq x$, and $f_D(f_D(x) = f_D(x)$.
- The range of f_D is a domain $(D_u, \sqsubseteq_D)$ such that
  - $(D_u, \sqsubseteq_D)$ and D are isomorphic,
  - $(D_u, \sqsubseteq_D)$ is a subdomain of U,
  - d is finite in D_u iff it is finite in U,
  - d is maximal in D_u iff it is maximal in U.

Since f_D is a projection we can even show that if $d_i \in D_u$, $d_1 \sqcup_U d_2 \in D_u$ and $d_1 \sqcap_U d_2 \in D_u$.

Even though ∇ provides a framework for defining embeddings that preserve necessary properties, it is not completely satisfactory solution to the problem of constructing a universal framework for higher order computation. ∇ suffers from two critical weaknesses. First, the effective presentation for function domains cannot be derived from effective presentations for the constituent domains. The effective presentations of A and B do not contain enough information about the global structure of A and B to determine which finite elements in $A \to_c B$ are maximal! Each function domain $A \to_c B$ must be separately embedded in ∇; the embedding cannot be constructed from embeddings for A and B.

The second weakness of ∇ is the unusual form of the universal language *Cons* provided with ∇. The language is a subset of a universal language for $\nabla^\top$. A legal program in *Cons* consists of a function definition F in the language for $\nabla^\top$ together with a proof that F maps inputs in ∇ to outputs in ∇. The reduction semantics for ∇ does not preserve the meaning of programs that violate this closure condition. It is conceivable that this condition could be enforced by a suitable syntactic type system, but Cartwright and Demers could not devise a plausible type system that would accept the functions required to show *Cons* is universal [CD88].

In this chapter, we present a new universal domain $\mathbf{T}$ with the following properties:

- The embeddings preserve maximality of finite elements.
- The universal language has a simple type checking system.
- All basic (ground) domains have effective presentations.
- An effective presentation for a function domain can be computed from the effective presentations of the constituent domains.

Our formulation of domain theory uses *OS*-functions, that is, the embeddings are performed by retractions that are *OS*-functions.

Even though the domain $\mathbf{T}$ preserves the finite maximality in its embeddings, it still does not preserve finiteness. We will fix this problem by constructing a refined universal domain $\mathbf{T}^*$ consisting of trees with infinite branching. We will present this solution at the end of this chapter.

5.2 Universal Domain of Lazy Binary Trees

The universal domain $\mathbf{T}$ is a domain of "lazy binary trees". It can be regarded as a generalization of S-expressions introduced in [M+65]. Unlike S-expressions, this domain accommodates both first-order and higher-order data.

These trees are referred to as lazy binary trees because the pairing operation does not evaluate its arguments. Two $\perp$'s paired together produces a non-$\perp$ *cons* node. This situation is analogous to the one discussed in [CD82]. In addition, these trees have binary branching. At any node the possible values are: natural number, errors, cons node. A cons node opens up two new leaves.

Trees have been used in the literature to construct universal domains such as $\mathcal{U}$ and ∇. However, since these universal domains need to embed domains with accommodate infinite descending chains, the trees are normalized by a quotient. This quotient normalizes infinite trees into finite elements. In our treatment of tree domain, we do not need to impose any quotient on the tree domain. It is a "natural" domain to do sequential computation. A computation proceeds by examining the tree in sequence. Each query on a tree translates to examining a particular node in the tree. Such a query may reveal the presence of an element at the node. All the possible values are in conflict with each other and thus belong to a C-set.

Definition 5.4. (*Universal Domain* $\mathbf{T}$) The universal domain $\mathbf{T}$ is a domain of lazy trees $(T, \sqsubseteq)$ where

$$T ::= \perp \mid \ulcorner n \urcorner \mid \mathsf{error}_1 \mid \mathsf{error}_2 \mid \langle T, T \rangle$$

and,

$$\begin{array}{rrcl} \forall x \in T & \perp & \sqsubseteq & x \\ \forall x \in T & x & \sqsubseteq & x \\ \forall x_1, y_1, x_2, y_2 \in T & \langle x_1, y_1 \rangle & \sqsubseteq & \langle x_2, y_2 \rangle \text{ iff } x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2. \end{array}$$

∎

This is clearly the generalization of B_t we introduced in Section 3.2.2. We now show that this domain is an OS-domain.

Proposition 5.5 The universal domain $\mathbf{T}$ is isomorphic to $\mathbf{D}(P_t)$ where $P_t = \langle P, \leq, C\rangle$ is defined as the following:

1. The set of primes P is given by the inductive definition:

$$P = \mathsf{error}_1 \mid \mathsf{error}_2 \mid \ulcorner n \urcorner \mid \langle \bot, \bot \rangle \mid \langle \bot, p \rangle \mid \langle p, \bot \rangle \qquad p \in P$$

2. The ordering relation $\leq$ on P is defined by the rules:

$$\begin{array}{rcll} x & \leq & x & \text{for} \quad x \in \mathbf{N} \cup \{\mathsf{error}_1, \mathsf{error}_2\} \\ \langle a, b\rangle & \leq & \langle c, d\rangle & \text{iff} \quad [(a = \bot) \vee (a \leq c)] \text{ and } [(b = \bot) \vee (b \leq d)] \end{array}$$

3. The *conflict* sets (C-sets) are defined by the equivalence closure of the relation C' over P:

$$\begin{array}{rcll} (0, \mathsf{error}_i),\ (0, \langle \bot, \bot \rangle) & \in & C' & \\ (0, n) & \in & C' & \text{for} \quad n \in \mathbf{N} \\ (\langle a, \bot\rangle, \langle b, \bot\rangle),\ (\langle \bot, a\rangle, \langle \bot, b\rangle) & \in & C' & \text{for} \quad (a, b) \in C' \end{array}$$

4. The two error elements in a C-set are the elements that have error_1 and error_2 at the end of the path.

It is easy to verify that this prime basis generates $\mathbf{T}$. Any downward closed, non-conflicting set represents a tree. In a tree at any node only one of the values, *i.e.*, the numbers and errors and a cons can be present. In gathering information about a value at a node, if one of the elements in a C-set is present, it negates the presence of all other elements in that set.

5.3 Subdomains

Dana Scott [Sco76]. introduced retractions as mechanisms to specify subdomains of complete *lattices*. In the context of complete lattices, he showed that a special class of retractions called closures was sufficient to define all subdomains. Plotkin generalized Scott's work to cpo's by introducing a larger class of retractions, called partial closures [Plo78], to identify subdomains.

Since our formulation of Domain Theory is based on OS-functions, we use partial closures that are observationally sequential. Since the universal language KL can express all the OS-functions over $\mathbf{T}$, the embedding function can be expressed in the universal language.

In this section we will show how to embed a given OS-domain D in $\mathbf{T}$ while preserving the maximality of finite elements. We will show that there is a computable OS-function partial closure f_D embedding D in $\mathbf{T}$. That means, the fixed point set of the function f_D is isomorphic to D.

Unlike the traditional domain theory, where the fixed point set of any partial closure is a domain, in $\mathbf{T}$, not all partial-closures produce OS-domains. [13] Since we can perform computation on any domain that can be embedded in the universal domain, it is possible to define OS-functions on a non-OS-domain provided it can be embedded in $\mathbf{T}$. However, we will not concern ourselves with such domains.

Consider the following example.

Example 5.1 Let f_D, a partial closure, be given with the following mapping.

$$\begin{aligned}
\perp &\longrightarrow \langle \perp, \perp \rangle \\
\mathsf{error}_i &\longrightarrow \langle \mathsf{error}_i, \mathsf{error}_i \rangle \\
\langle \perp, \perp \rangle &\longrightarrow \langle \perp, \perp \rangle \\
\langle \perp, \mathsf{error}_i \rangle &\longrightarrow \langle \perp, \mathsf{error}_i \rangle \\
\langle \mathsf{error}_i, \perp \rangle &\longrightarrow \langle \mathsf{error}_i, \perp \rangle \\
\langle 1, \perp \rangle &\longrightarrow \langle 1, \perp \rangle \\
\langle \perp, 3 \rangle &\longrightarrow \langle \perp, 3 \rangle
\end{aligned}$$

[13] It may be possible to restrict partial closures to produce only OS-domains. That topic is an open question.

$$\langle 2, \mathsf{error}_i\rangle \longrightarrow \langle \mathsf{error}_i, \mathsf{error}_i\rangle$$
$$\langle \mathsf{error}_i, 2\rangle \longrightarrow \langle \mathsf{error}_i, \mathsf{error}_i\rangle$$
$$\langle 2, 2\rangle \longrightarrow \langle 2, 2\rangle$$

It can be easily seen that this function is a partial closure and an OS-function. Therefore, its range is a subdomain of $\mathbf{T}$. However, it is not an OS-domain, because the elements above $\langle\perp,\perp\rangle$ do not form C-sets. In particular, the element $\langle 2,2\rangle$ is in conflict with $\langle 1,\perp\rangle$ and $\langle\perp,3\rangle$, but the elements $\langle 1,\perp\rangle$, $\langle\perp,3\rangle$ are not in conflict with each other. ∎

5.3.1 Presentation of OS-domains

The domain $\mathbf{T}$ is universal for the class of OS-domains. That is, any OS-domain can be embedded in $\mathbf{T}$ by an OS-function. In addition, the embedding preserves the maximality of finite elements. Since the embedding process produces error elements, the domains that need to be embedded must be error-rich.

A computation in a domain D is an enumeration producing finite approximations to the answer with progressively higher accuracy. A computation terminates if and only if the answer is a finite maximal element. To preserve the behavior of computations over D, an embedding of D in $\mathbf{T}$ must map the finite elements in terminating computations to finite elements of $\mathbf{T}$ and their answers to finite maximal elements of $\mathbf{T}$.

To satisfy this property, embeddings must preserve more structure than the embeddings used in classical domain theory [Plo78, Sco76], which do not preserve finite maximality. To describe the embedding process and make it *effective*, we need to define what we mean by an *effective presentation* of a domain.

Since our domains are constructed from prime elements and our embedding must preserve finite maximality, our definition of effective presentation differs from others in the literature:

Definition 5.6. (*Effective Presentation*) An effective presentation $\mathcal{E}$ of the domain $\mathbf{D}(P, \leq, C)$ is a tuple $\langle \mathbf{E}, \mathbf{c}, \mathbf{pr}, \mathbf{arity}, \mathbf{size} \rangle$ such that:

1. $\mathbf{E} : P \rightarrow \mathbf{N}$ is an injective function that enumerates the prime elements of the domain excluding errors in topologically sorted order. That is, if an element a approximates b, a is enumerated before b. The error elements are not enumerated because their presence is implied by the fact that $\mathbf{D}(P, \leq, C)$ is an OS-domain. We use the notation p_i for $p \in P$ when $\mathbf{E}(p) = i$. We also use the convention $\mathbf{E}(0) = \perp$.

2. $\mathbf{c} : \mathit{range}(\mathbf{E}) \rightarrow \mathbf{N}$ is a function that maps the index of the prime element to its C-set code. That is, $\forall i, j > 0 : \mathbf{c}(i) = \mathbf{c}(j)$ iff $p_i, p_j \in Q$ for $Q \in C$. The range of the function $\mathbf{c}$ is the set of C-set numbers.

3. $\mathbf{pr}$ is a partial recursive function over $\mathbf{N}$ defined by $\mathbf{pr}(i) = j$ iff $p_i \prec p_j$.

4. $\mathbf{arity}$ is a partial recursive function that maps the index of a prime element, including 0, to the number in $\mathbf{N} \cup \{\omega\}$ of C-sets above that element;[14] and

5. $\mathbf{size}$ is a partial recursive function that maps the index of a C-set to its cardinality in $\mathbf{N} \cup \{\omega\}$.

∎

All flat domains such as numbers, booleans, and non-lazy lists obviously have effective presentations. Moreover the domain constructions like cartesian product, disjoint union, lifting, and the OS-function construction preserve the effective presentation. As a result, all the domains typically encountered in deterministic sequential computation can effectively be embedded in $\mathbf{T}$ while preserving the maximality of finite elements.

Example 5.2 The domain $\mathbf{N}_{\perp}^{E}$ can be effectively be presented as follows.

[14] A prime element may have ω C-sets above it.

- $\mathbf{E}(n) = n + 1$.
- $\mathbf{c}(n) = 0$.
- $\mathbf{pr}(n) = 0$.
- $\mathbf{arity}(0) = 1$ and $\mathbf{arity}(n+1) = 0$.
- $\mathbf{size}(0) = \omega$.

■

Proposition 5.7 If the OS-domains D_1 and D_2 are effectively presented, so are the domains $D_1 \times D_2$, $D_1 \oplus D_2$, and $\odot(D)$.

Proof. Let $\mathcal{E}_1, \mathcal{E}_2$ be the effective presentations of D_1 and D_2.

- $\mathcal{E}_\times$: We can define the effective presentation $\mathcal{E}_\times = \langle \mathbf{E}, \mathbf{c}, \mathbf{pr}, \mathbf{arity}, \mathbf{size} \rangle$ as follows.

  - $\mathbf{E}$: We know that all the primes of $D_1 \times D_2$ are of the form $\langle p_1, \perp \rangle$ or $\langle \perp, p_2 \rangle$. We define

    $$\begin{aligned} \mathbf{E}(\langle p_1, \perp \rangle) &= \langle 0, \mathbf{E}_1(p_1) \rangle \\ \mathbf{E}(\langle \perp, p_2 \rangle) &= \langle 1, \mathbf{E}_2(p_2) \rangle \end{aligned}$$

    where $\langle \cdot, \cdot \rangle : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ is the standard pairing function defined by

    $$\langle i, j \rangle = \frac{(i+j) * (i+j+1)}{2} + i\,.$$

    It is obvious that $\mathbf{E}$ is one-to-one function. Also, $\mathbf{E}$ does not assign 0 to any element keeping it free for $\perp$.

  - $\mathbf{c}$: We define this function as follows.

    $$\begin{aligned} \mathbf{c}(\langle 0, i \rangle) &= \langle 0, \mathbf{c}_1(i) \rangle \\ \mathbf{c}(\langle 1, j \rangle) &= \langle 1, \mathbf{c}_2(j) \rangle \end{aligned}$$

- **pr** :

$$\begin{aligned}\mathbf{pr}(\langle 0,i\rangle) &= \langle 0,\mathbf{pr}_1(i)\rangle\\ \mathbf{pr}(\langle 1,j\rangle) &= \langle 1,\mathbf{pr}_1(j)\rangle\end{aligned}$$

whenever $\mathbf{pr}_1$ and $\mathbf{pr}_2$ are defined on i and j respectively.

- **arity** :

$$\begin{aligned}\mathbf{arity}(\langle 0,i\rangle) &= \mathbf{arity}_1(i)\\ \mathbf{arity}(\langle 1,j\rangle) &= \mathbf{arity}_2(j)\\ \mathbf{arity}(0) &= \mathbf{arity}_1(0)+\mathbf{arity}_2(0)\end{aligned}$$

Notice that $\perp$ of the product space has an arity that equals the sum of arities of $\perp$'s of the constituent spaces.

- **size** :

$$\begin{aligned}\mathbf{size}(\langle 0,i\rangle) &= \mathbf{size}_1(i)\\ \mathbf{size}(\langle 1,j\rangle) &= \mathbf{size}_2(j)\end{aligned}$$

- $\mathcal{E}_\oplus$ The effective presentation for the disjoint sum of two domains $\langle \mathbf{E},\mathbf{c},\mathbf{pr},\mathbf{arity},\mathbf{size}\rangle$ can be defined as follows.

  - **E**: Since D_1+D_2 has two new elements, we assign the indices as follows:

  $$\begin{aligned}\mathbf{E}(\langle x,1\rangle &= \langle 1,\mathbf{E}_1(x)\rangle\\ \mathbf{E}(\langle y,2\rangle &= \langle 2,\mathbf{E}_2(x)\rangle\end{aligned}$$

  Notice that this scheme assigns $\langle 1,0\rangle$ and $\langle 2,0\rangle$ to $\langle\perp,1\rangle$ and $\langle\perp,2\rangle$ leaving 0 free for $\perp$.

  - **c**:

  $$\begin{aligned}\mathbf{c}(\langle j,i\rangle) &= \langle j,\mathbf{c}_j(i)\rangle \text{ for } j=1,2\\ \mathbf{c}(\langle j,0\rangle) &= 0 \text{ for } j=1,2\end{aligned}$$

  That is, the C-set number for the additional C-set, $\{\mathsf{error}_1,\mathsf{error}_2,\langle\perp,1\rangle,\langle\perp,2\rangle\}$ is 0. The other C-sets are prefixed with the domain number.

  - **pr**:

  $$\begin{aligned}\mathbf{pr}(\langle j,0\rangle) &= 0 \text{ for } j=1,2\\ \mathbf{pr}(\langle j,i\rangle) &= \langle j,\mathbf{pr}(i)\rangle \text{ for } j=1,2\end{aligned}$$

That is, the newly added $\perp$'s are immediately above $\perp$ of the disjoint sum.

- **arity** :

$$\begin{aligned}\textbf{arity}(\langle j,i\rangle) &= \textbf{arity}_j(i) \text{ for } j=1,2\\ \textbf{arity}(0) &= 1\end{aligned}$$

That is, there is only one C-set above $\perp$ consisting only of the newly added $\perp$'s and the elements error_1 and error_2.

- **size**:

$$\begin{aligned}\textbf{size}(0) &= 2\\ \textbf{size}(\langle j,i\rangle) &= \textbf{size}_j(i) \text{ for } j=1,2\end{aligned}$$

$\square$

Since most commonly used domain constructions preserve the effectiveness of presentations, it is possible to embed the domains constructed from the ground domains in $\mathbf{T}$ effectively. If a programming language includes functions as higher-order data, that data domain is formed by suitable function construction on other domains. If we want to embed the function spaces effectively, we must be able to derive the effective presentation from the effective presentations of the constituent domains. In the following paragraphs, we derive the effective presentation of the OS-function space. This task requires us to index the *finite* elements, since finite elements from the constituent domains form an important part of the function space construction.

Encoding Finite Elements

Since finite elements are least upper bounds of finite sets of primes, it is easy to generate codes for the finite elements of D given $\mathbf{E}$, the enumeration of the prime elements. The index of a finite element d is the integer code representing the finite set S_d of prime indices of primes below d. To make this encoding unique, we represent S_d as a sorted list of prime indices terminated by the *end-of-list* marker 0 and use a standard bijective pairing function to encode this list as an integer.

Definition 5.8. (*Index of finite element*) In the *OS*-domain with prime enumerator $\mathbf{E}$, the finite element $a = \sqcup\{p_{k_1}, p_{k_2}, \ldots, p_{k_n}\}$ where $k_1 > k_2 > \ldots > k_n$ has the *index*

$$\langle k_1, \langle k_2, \ldots, \langle k_n, 0\rangle \ldots\rangle\rangle$$

where $\langle\cdot,\cdot\rangle : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ is the standard pairing function defined previously. We denote the finite element with index I by a_I. We superscript the element by the domain it belongs to, if there is ambiguity.

Note: Each prime element has two indices: one as a prime element and another as a finite element. ∎

The encoding described is unique for every n-tuple of positive numbers $\langle k_1, \ldots, k_n\rangle$. Since the pairing function, the projection functions, and the sorting function over the encoded tuples are partial recursive, they can be expressed in the universal language.

Proposition 5.9 If the effective presentations for domains D_1 and D_2 are given, we can compute the effective presentation for the domain $D_1 \rightarrow_{os} D_2$.

Proof. Let the indices I, J range over the indices of finite elements and i, j, k range over the indices of prime elements. We refer to the finite element with index I as a_I and the prime element with index i as p_i. We superscript elements with their domains whenever it is necessary to avoid confusion.

In the presentation, we concentrate on providing a suitable encoding for prime elements that makes it easy to express other functions such as $\mathbf{c}$. Hence, the encodings are more involved than those in $\times, +$.

- $\mathbf{E}$: We inductively define the indexing as follows.

$$\begin{array}{rcl} \mathbf{E}(\perp \rightarrow_{os} \perp) & = & 0 \\ \mathbf{E}(f) & = & \begin{cases} \langle\langle 0, I, k\rangle, l\rangle & \text{if } f \in a_I \xrightarrow{o} p_k \text{ and } f \succ f_l. \\ \langle\langle 1, \langle I, m\rangle, n\rangle, l\rangle & \text{if } f \in a_I : C_n \xrightarrow{s} C_m \text{ and } f \succ f_l. \end{cases} \end{array}$$

  where $\mathbf{E}(f_l) = l$. It is clear from the above encoding that the enumeration is topologically sorted. Since the domains satisfy the property **[UP]**, such topologically sorted presentation is possible.

- $\mathbf{c}$:

$$\begin{aligned}\mathbf{c}(\langle\langle 0, I, k\rangle, l\rangle) &= \langle\langle I, \mathbf{c}_2(k)\rangle, l\rangle \\ \mathbf{c}(\langle\langle 1, \langle I, m\rangle, n\rangle, l\rangle) &= \langle\langle I, m\rangle, l\rangle\end{aligned}$$

From the above encoding, it is easy to show that elements in the same C-set are mapped to the same number.

- **pr** : This function selects the second component of a tuple $\langle i, j\rangle$ since p_j is the immediate predecessor of the prime element $p_{\langle i,j\rangle}$.

- **arity**: The function **arity** is defined by the equations:

$$\begin{aligned}\mathbf{arity}(0) &= \mathbf{arity}_2(0) \\ \mathbf{arity}(\langle\langle 0, I, k\rangle, l\rangle) &= \mathbf{arity}_2(k) \\ \mathbf{arity}(\langle\langle 1, \langle I, m\rangle, n\rangle, l\rangle) &= \mathbf{size}_1(n)\end{aligned}$$

The number of C-sets above an output prime f depends on the number of C-sets above the final output p_k of f. At that point, the function f may generate incremental outputs in all the directions above p_k; therefore $\mathbf{arity}(f) = \mathbf{arity}_2(k)$. If $f \in a_I : C_n \xrightarrow{s} C_m$, f may explore a C-set for each element in the input C-set C_n; therefore $\mathbf{arity}(f) = \mathbf{size}_1(n)$. We will illustrate this point with an example later.

- **size**: Let the index of the C-set Q be $\langle\langle I, m\rangle, l\rangle$. Let S_I be the set of indices primes on the "fringe" of a_I, *i.e.*, if $i \in S_I$ then $p_i \in a_I$ and there is no $p_j \in a_I$ such that $\mathbf{pr}(j) = i$. Then,

$$\mathbf{size}(\langle\langle I, m\rangle, l\rangle) = \mathbf{size}_2(m) + \sum_{i \in S_I} \mathbf{arity}_1(i)$$

where the addition operator $+$ has been extended to include ω. We can easily verify that the function **size** gives the cardinality of the C-set in $D_1 \rightarrow_{os} D_2$. It is clear that the number of output primes in the C-set Q is exactly the number of primes in the output C-set C_m. The C-set Q has a schedule prime for each direction above a_I. Thus, the preceding definition of **size** computes the number of elements in the C-set.

□

Here is an example for an effective presentation for a function space.

Example 5.3 Consider the domain $\mathbf{B}_\perp^E$, the domain of booleans enriched with two errors. Let $\mathbf{B}$ be $\{\mathsf{tt}, \mathsf{ff}\}$ and $\mathbf{B}^E$ be $\{\mathsf{error}_1, \mathsf{error}_2\} \cup \mathbf{B}$. The effective presentation of $\mathbf{B}_\perp^E$ is given as follows:

E: $p_0 = \perp, p_1 = \mathsf{tt}, p_2 = \mathsf{ff}$.

c: $\mathbf{c}(p_1) = \mathbf{c}(p_2) = 0$.

pr: $\mathbf{pr}(p_1) = \mathbf{pr}(p_2) = 0$.

arity: $\mathbf{arity}(0) = 1, \mathbf{arity}(1) = \mathbf{arity}(2) = 0$.

size: $\mathbf{size}(0) = 2$.

The domain $\mathbf{B}_\perp^E \rightarrow_{os} \mathbf{B}_\perp^E$ has the following functions.

Constant primes generate output independent of their input. They are, for $e \in \{\mathsf{tt}, \mathsf{ff}, \mathsf{error}_1, \mathsf{error}_2\}$,

$$g_e = \perp \rightarrow e$$

The diverging prime s inspects its input and diverges. It is represented by

$$s = \{\perp \rightarrow \perp, \mathsf{error}_1 \rightarrow \mathsf{error}_1, \mathsf{error}_2 \rightarrow \mathsf{error}_2\}$$

Strict primes output the designated elements if the input has enough information. Since these functions evaluate the input they generate error when they encounter error elements. For every pair $d \in \mathbf{B}$ and $e \in \mathbf{B}^E$, there is a strict prime:

$$f_{d,e} = \{d \rightarrow e, \mathsf{error}_1 \rightarrow \mathsf{error}_1, \mathsf{error}_2 \rightarrow \mathsf{error}_2\}$$

Thus, the prime basis for $\mathbf{B}_\perp^E \rightarrow_{os} \mathbf{B}_\perp^E$ can be described as follows:

$P \;\; = \{g_e, s, f_{d,e} \,|\, d \in \mathbf{B}, e \in \mathbf{B}^E\}$

$C \;\;$ is identified by the C-sets S, T_d for $d \in \mathbf{B}$ where

$$\begin{aligned} S &\stackrel{df}{=} \{s, g_e \,|\, e \in \mathbf{B}^E\} \\ T_d &\stackrel{df}{=} \{f_{d,e} \,|\, e \in \mathbf{B}^E\} \end{aligned}$$

The designated error elements are g_{error_1} and g_{error_2} for S and f_{d,error_1} and f_{d,error_2} for T_d.

Now we will present this domain as follows:

$\mathbf{E}_\rightarrow$: The enumeration is given by the equations:

$$\begin{aligned} \mathbf{E}_\rightarrow(\perp \rightarrow_{os} \perp) &= 0 & \\ \mathbf{E}_\rightarrow(g_e) &= \langle\langle 0, 0, \mathbf{E}(e)\rangle, 0\rangle & \text{for } e \in \mathbf{B} \\ \mathbf{E}_\rightarrow(s) &= \langle\langle 1, \langle 0, 0\rangle, 0\rangle, 0\rangle & \\ \mathbf{E}_\rightarrow(f_{d,e}) &= \langle\langle 0, \mathbf{E}_{finite}(d), \mathbf{E}(e)\rangle, \mathbf{E}_\rightarrow(s)\rangle & \text{for } e \in \mathbf{B} \end{aligned}$$

Notice that the error functions are not given any indices since they are implicitly present. Here, $\mathbf{E}_{finite}$ is a function that assigns the indices for finite elements as described above.

$\mathbf{c}_\rightarrow$: It can easily be verified that $\mathbf{c}_\rightarrow$ for the elements of S is

$$\langle\langle 0, 0\rangle, 0\rangle$$

since $\mathbf{c}(\mathbf{E}(e))$ for $e \in \mathbf{B}$ is 0. Also, $\mathbf{c}_\rightarrow$ for the elements of any T_d is

$$\langle\langle \mathbf{E}_{finite}(d), 0\rangle, \mathbf{E}_\rightarrow(s)\rangle$$

thus satisfying the conditions of C-set numbering.

$\mathbf{pr}_\rightarrow$: It can easily be verified that $\mathbf{pr}_\rightarrow(f) = 0$ for all $f \in S$ and $\mathbf{pr}_\rightarrow(f) = \mathbf{E}_\rightarrow(s)$ for all $f \in T_d$ for any $d \in \mathbf{B}$.

arity$_\rightarrow$: Consider the arity of $\perp \rightarrow_{os} \perp$. It has only one C-set above it. Since it can be considered an output prime with final output as $\perp$, there is only one C-set can contribute to the increase of the output, namely the C-set $\mathbf{B}$. Hence, it has **arity** 1. By our description it is the arity of element 0, *i.e.*, which by the formula given by $\mathbf{arity}(0)$.

Similarly, the **arity** of g_e is given by $\mathbf{arity}(\mathbf{E}(e))$, which is 0 as expected. Also, the **arity** of s is given by the equation $\mathbf{size}(0)$, which is 2. The different direction that s can explore further for each element in the C-set $\mathbf{B}$ gives this number. The two C-sets above s are T_{tt} and T_{ff}. The formula gives the same number.

The **arity** for any $f_{d,e}$ is given by $\mathbf{arity}(\mathbf{E}(e))$ for $e \in \mathbf{B}$, which is 0. Therefore these primes are maximal primes, which has been shown earlier.

size: For the elements of the C-set S, the **size** according to the formula is

$$\mathbf{size}(\langle\langle 0,0\rangle,0\rangle) = \mathbf{size}_2(0) + \sum_{i\in S_I} \mathbf{arity}_1(i) = \mathbf{size}(0) + \mathbf{arity}(0) = 2+1$$

Here S_I is the elements on the fringe of $\perp$ which is only $\perp$. Since the number of elements of S is 3, with two output primes and one schedule prime, it confirms the formula.

The size of any T_d can be calculated as

$$\mathbf{size}(0) + \sum_{i\in S_I} \mathbf{arity}_1(i) = \mathbf{size}(0) + \mathbf{arity}(\mathbf{E}(d)) = 2.$$

Since the **size** of the C-set 0 is two, there can be two output elements, one for each element in the C-set 0. Since the arity of tt as well as ff is 0, there can be no schedule primes conflicting with any element of T_d.

Notice that the error elements are excluded when counting the **size**.

■

5.3.2 Embedding a Subdomain

We can embed a domain D with effective presentation $\mathcal{E}$ by computing the subdomain in $\mathbf{T}$. In this section, we compute the embedding for a domain with a given presentation. We will also show that the subdomain we identified defines an observably sequential retraction whose range is the subdomain.

A domain D is embedded in $\mathbf{T}$ by a function f_d mapping its prime elements into $\mathbf{T}$. For each prime p, $f_d(p)$ is computed by inserting a subtree into the tree corresponding to the predecessor of p. Since all the prime elements are enumerated in topologically sorted order, the embedding can be defined inductively. An arbitrary element of the domain D can be embedded by embedding all its prime approximations.

Since the effective presentation of D does not include the error elements, we do not provide an embedding for them. That is, the subdomain we identify is isomorphic to D without errors. However, when we provide the corresponding observably sequential retraction, its fixed point set is isomorphic to D. In addition, the error elements of D are mapped to finite elements containing error elements in $\mathbf{T}$.

Let D be the domain effectively given by the enumeration $\mathcal{E}$. We will first identify the subset of the domain $\mathbf{T}$ isomorphic to D without errors and later show this to be an effective subdomain.

For convenience, we use $\langle a, b, c, \ldots \rangle$ for $\langle a, \langle b, \langle c, \ldots \rangle \rangle \rangle$.

Induction Hypothesis: The function f maps the prime elements into $\mathbf{T}$ such that the following properties are satisfied.

1. If $(p, q) \in C$, then $f(p) \# f(q)$ in $\mathbf{T}$. That is, conflicting primes are mapped to inconsistent finite elements.

2. If $p, q \in a$ where a is a finite element of D, that is, p, q can form a finite-prime ideal, then $f(p) \uparrow f(q)$.

3. If $p \leq q$, then $f(p) \sqsubseteq f(q)$.

4. For every p_i, $f(p_i)$ contains a specially marked subtree s_i such that s_i has exactly the same number of $\perp$'s as the number of C-sets above p_i.

Base Case: f maps p_0, which must be $\perp$, to $\perp^l$ where l is the number of C-sets above $\perp$. The difference between l and **arity** is that **arity** must represent ω in $\mathbf{N}$, where as l could be ω. For example, if **arity** represents ω as 0 and n as $n+1$, we would have the following scenario. Let **arity**(0) be 2, *i.e.*, represented as 3. Then the subtree would be $\langle \perp, \perp \rangle$. If the **arity** is ω, the subtree is $\langle \perp, \perp, \perp, \ldots \rangle$. That means, there are exactly as many number of $\perp$'s as there are C-sets above p_0. The subtree s_0 is the entire tree. Thus, the induction hypothesis can be verified.

Induction Step: Assume that we have embedded all the primes in the enumeration from p_0 through p_{i-1}. We define $f(p_i)$ as follows:

Let k be the cardinality of the set of the already enumerated elements that are in conflict with p_i. That is, k prime elements belonging to the C-set of p_i have been enumerated so far. Define s_i, a subtree for p_i as $\langle k, \perp^l \rangle$ where l is the number of C-sets above p_i. If $l = 0$, then define it as k.

Let q be the element immediately below p_i. By induction, q has been embedded in $\mathbf{T}$ as a tree $f(q)$. The element q was embedded in $\mathbf{T}$ by adding a subtree s_q to the image of q's predecessor. In the subtree s_q, insert s_i (the subtree corresponding to p_i) at position $g(p_i)$, where the function $g(p_i)$ is defined as follows. If a prime p' belonging to the same C-set as p_i has been enumerated, then $g(p_i) = g(p')$. Otherwise, $g(p_i)$ is the number of distinct C-sets above q that are already enumerated. By the induction hypothesis, the position $g(p_i)$ has $\perp$ in the subtree s_q.

We will verify that the construction satisfies the induction hypothesis.

1. *If some p_j is in conflict with p_i then $f(q)\#f(p_i)$.* Let q be the prime (or $\perp$) immediately below p_i and p_j. Since p_j is in conflict with p_i, by construction they would have been embedded in the same position in s_q where s_q is the subtree for q. Also, since the number of elements in the same C-set enumerated before p_j and before p_i are different, s_i and s_j are inconsistent. Therefore, $f(p_i)\#f(p_j)$.

2. *If p_k is not in conflict with p_i, then $f(p_k) \uparrow f(p_i)$.* If p_k and p_i are not in conflict, then p_k is not in conflict with all the primes below p_i. By IH, all those primes are mapped to trees consistent with p_k. If p_k is above q belonging to a different C-set, then s_k and s_i are inserted at different places in s_q. Thus, they are consistent trees.

3. *If $p_j \sqsubseteq p_i$, then $f(p_j) \sqsubseteq f(p_i)$.* The tree for p_i certainly dominates the tree for q, the immediate predecessor. Hence, by IH, $f(p_j) \sqsubseteq f(p_i)$.

4. We have already marked a special tree s_i for p_i with as many $\perp$'s as there are C-sets above p_i.

Now, we have a mapping from prime elements, including $\perp$, to the finite elements of the tree domain. The finite elements of D can be mapped as the least upper bounds of the mappings of their prime approximations. We can show the following important property of the embeddings.

Proposition 5.10 The preceding mapping maps a finite maximal element of D to a finite maximal tree.

Proof. Consider a finite maximal element $a \in D$. The elements in the fringe must have **arity** 0. That is, there are no C-sets above the prime elements in the fringe. Hence, the corresponding subtrees in the embeddings do not have $\perp$'s.

Any prime element p not in the fringe must have $\perp$'s in its subtree. Since a is maximal, a must contain one prime from each C-set above p. Thus, all the $\perp$'s in the subtree for p are filled in the mapping for a. Therefore, the mapping of a does not contain any $\perp$'s, hence, a maximal tree.

Since no prime element in a has infinite C-sets above it, the mappings of all prime elements are finite. Since a contains only a finite number of prime approximations, the mapping for a is a finite tree.

Hence a is mapped to finite maximal tree.

$\square$

Therefore, we have identified a subset R of the universal domain $\mathbf{T}$ with the following properties:

- If $r, s \in R$ and $r \sqcup s \in \mathbf{T}$ then $r \sqcup s \in R$. The $\sqcup$ operation is from the domain $\mathbf{T}$.
- If $r, s \in R$ and $r \sqcup s \in R$ then $r \sqcap s \in R$.
- Let P_r be the subset of R, which are the mappings of prime elements. Hence they cannot be described as lubs of other finite elements. Then, the set of primes under $\sqsubseteq$ ordering and $\#$ relation forms a prime basis.

Given those properties of the set R, we can construct a function f with fixed point set R' where R' is R enriched with error elements.

We construct the function f inductively as follows. Let R_n be the subset of R such that the primes of R_n are at distance at most n, from r_0 where r_0 is the least element in R.

Induction Hypothesis: The OS-function f over $\mathbf{T}$ has the fixed point R_n with two error elements in each C-set.

Base Case: Let r_0 be the minimal element of R. Then $R_0 = \{r_0\}$. Define the function f as:

$$f(\perp) = r_0$$

This function has the fixed point set R_0 and is an OS-function.

Induction Step: Let the fixed point set of f be error-enriched R_n. We extend f to include R_{n+1} as follows.

Let Q be a set of primes in R just above a prime element r. This element need not be a prime in $\mathbf{T}$, but it is a prime element in R. By IH, $f(r) = r$. Let r' be the least element such that $f(r) = r$.

From the construction of R we know that there is a position in r where all the elements of Q are situated. That is, there is a position t in r where there is $\perp$ and all the elements of Q are above r at that position. In fact, each element q_i in Q has a subtree $s_i = \langle i, \perp^l \rangle$ where l is the arity of q_i such that $r[t/s_i] = q_i$. We use the notation $r[t/s]$ for the tree that results from substituting s at position t. We extend the function to include:

$$
\begin{aligned}
f(r[t/\langle i, \perp\rangle]) &= q_i \text{ for } q_i \in Q \wedge s_i = \langle i, \perp^l \rangle \\
f(r[t/j] &= q_j \text{ for } q_j \in Q \wedge s_j = j
\end{aligned}
$$

In addition, since $f(r') = r$, the function must search for s_i in the tree. If it encounters an error element along the way, it must generate an error element. We capture this error-sensitive behavior as follows.

Let S_i be the set of primes in $q_i' - r'$. That is, q_i' is the minimal element mapped to q_i. Let S_i^E be the set of error primes for each element in S_i. Since f is error-sensitive, f maps these error elements into an error element. For all error_1^j and error_2^j in S_i^E define f as the minimal function satisfying:

$$r[t/\mathsf{error}_1] \in f(r' \sqcup \mathsf{error}_1^j)$$

$$r[t/\mathsf{error}_2] \in f(r' \sqcup \mathsf{error}_2^j)$$

It can easily be verified that f is observably sequential. By minimal, continuous extension, f is defined on all the finite elements and all the finite elements of R are in its fixed point set.

In the preceding extension of f, each C-set Q is has two error elements, for example, $r[t/\mathsf{error}_1]$, $r[t/\mathsf{error}_2]$. Thus, the fixed point set of the function f is an OS-domain that is isomorphic to the original domain including the error elements.

Therefore, given a domain D with an effective presentation, we can construct a partial closure whose fixed point set is isomorphic to D. In addition, if an element in D is finite maximal, its image in $\mathbf{T}$ is finite maximal as well. Let us consider some examples of embeddings.

Example 5.4 Consider the domain $\mathbf{N}_{\perp}^{E}$. As per the preceding construction, we embed $\perp$ as $\perp$. If the enumeration assigns n to each number n, the subtree for each natural number is n. Hence, the subdomain consists of $\perp$ and $n \in \mathbf{N}$. The embedding function is I: $\{(\perp,\perp),(\mathsf{error}_i,\mathsf{error}_i),(n,n) \mid n \in \mathbf{N}\}$. ∎

Example 5.5 Consider the domain $\mathbf{B}_{\perp}^{E} \rightarrow_{os} \mathbf{B}_{\perp}^{E}$ outlined in the example 5.3. This domain can be embedded as follows.
Since the arity of the least defined function $\perp \rightarrow_{os} \perp$ is 1, we can embed it as:

$$(\perp \rightarrow_{os} \perp) \longrightarrow \perp$$

Each element in S, *i.e.*, the element $g_e = \perp \rightarrow_{os} e$ and $s = \mathsf{error}_i \rightarrow_{os} \mathsf{error}_i$, is mapped as follows. Assume that the enumeration goes in the order s, g_{tt}, and g_{ff}.

$$\begin{array}{rcll} s & \longrightarrow & \langle 0,\perp,\perp\rangle & \text{since } \mathbf{arity}(s)=2. \\ g_{\mathsf{tt}} & \longrightarrow & 1 & \text{since } \mathbf{arity}(g_{\mathsf{tt}})=0 \\ g_{\mathsf{ff}} & \longrightarrow & 2 & \text{since } \mathbf{arity}(g_{\mathsf{ff}})=0 \end{array}$$

The functions above s which are $f_{d,e}$ are mapped as follows:

$$\begin{array}{rcll} g_{\mathsf{tt},\mathsf{tt}} & \longrightarrow & \langle 0,0,\perp\rangle & \text{Since the subtree is 0} \\ g_{\mathsf{tt},\mathsf{ff}} & \longrightarrow & \langle 0,1,\perp\rangle & \text{Since the subtree is 1} \\ g_{\mathsf{ff},\mathsf{tt}} & \longrightarrow & \langle 0,\perp,0\rangle & \text{Since the subtree is 0} \\ g_{\mathsf{tt},\mathsf{tt}} & \longrightarrow & \langle 0,\perp,1\rangle & \text{Since the subtree is 1} \end{array}$$

The corresponding function is:

$$\begin{array}{rcl@{\qquad}rcl}
\perp & \rightarrow & \perp & \mathsf{error}_i & \rightarrow & \mathsf{error}_i \\
1 & \rightarrow & 1 & 2 & \rightarrow & 2 \\
\langle \mathsf{error}_i, \perp \rangle & \rightarrow & \mathsf{error}_i & \langle 0, \perp \rangle & \rightarrow & \langle 0, \perp, \perp \rangle \\
\langle 0, \mathsf{error}_i \rangle & \rightarrow & \langle 0, \mathsf{error}_i, \mathsf{error}_i \rangle & \langle 0, \mathsf{error}_i, \perp \rangle & \rightarrow & \langle 0, \mathsf{error}_i, \perp \rangle \\
\langle 0, 0, \perp \rangle & \rightarrow & \langle 0, 0, \perp \rangle & \langle 0, 1, \perp \rangle & \rightarrow & \langle 0, 1, \perp \rangle \\
\langle 0, \perp, \mathsf{error}_i \rangle & \rightarrow & \langle 0, \perp, \mathsf{error}_i \rangle & & & \\
\langle 0, \perp, 0 \rangle & \rightarrow & \langle 0, \perp, 0 \rangle & \langle 0, \perp, 1 \rangle & \rightarrow & \langle 0, \perp, 1 \rangle
\end{array}$$

The range of the preceding retraction is isomorphic to $\mathbf{B}_{\perp}^{E} \rightarrow_{os} \mathbf{B}_{\perp}^{E}$ including all the error elements. ∎

In the preceding examples, the finite elements of the domain D are mapped to finite trees. In general, a finite element may be mapped to an infinite element. Consider the element Z$\perp$ in T^{ω}. This element has an infinite number of C-sets above it. Hence, it is mapped to $\perp^{\omega}$, which is not a finite tree.

Computationally, it is more meaningful to have finite elements mapped to finite elements. Given a domain D embedded in the universal domain, the embedding must be sufficient to carry out computations on the original domain. If a finite element d of D is mapped to an infinite tree, the computation can never produce the finite element. If the computation is carried out on the original domain, it is possible to see the intermediate result d, since it is finite. The universal domain can never generate sufficient output to denote d as it is an infinite tree.

Example 5.6 Consider T^{ω}, the infinite product of the $\mathsf{tt}, \mathsf{ff}, \perp$. The element $\langle \perp, \perp \ldots \rangle$ is the minimal element, *i.e.*, $\perp$ of the domain T^{ω}. It is easy to see that $\perp$ is mapped to $\perp^{\omega}$ and $\langle \mathsf{tt}, \perp, \ldots \rangle$ is mapped to $\langle 0, \perp^{\omega} \rangle$.

If the computation has to generate the corresponding tree to $\langle \mathsf{tt}, \perp, \perp, \ldots \rangle$, it must generate an infinite tree with the first branch 0. Since, this process can never terminate, we may never see a finite element in the output.

However, in the embedding, each finite element of T^{ω} mapped to an infinite tree has a finite synonym. That is, all the synonyms are coerced to the same infinite element. For example, $\langle 0, \perp \rangle$, which is a finite tree, is mapped to the infinite tree $\langle 0, \perp^{\omega} \rangle$. Since it is decidable to recognize the existence of these synonyms in the output, it is computably tractable to check for finite elements in the output. ∎

We can provide another universal domain where finite elements are mapped to finite elements. It requires domains with infinite branching.

5.4 Universal Domain of Trees with Infinite Branching

The universal domain $\mathbf{T}$ has finite maximal elements that are needed to embed the domains faithfully. However, it lacks a certain kind of finite elements, namely those that have infinite number of C-sets dominating it. The absence of such elements prevents the preservation of finiteness of the embedded domains.

The domain $\mathbf{T}^*$ solves the problem by generalizing the *cons* node. A leaf can be a node with 2, 3, or even ∞ branches. That is, these nodes open up a list of $\perp$'s that can be potentially infinite. Each cons node with different arity will have a different label. Thus, a cons node with label 2 is in conflict with a cons node with label 3.

The domain $\mathbf{T}^*$ is defined as follows.

Definition 5.11. (*Universal Domain* $\mathbf{T}^*$) The universal domain $\mathbf{T}^*$ is a domain of lazy trees $(T^*, \sqsubseteq)$ where T^* and $\sqsubseteq$ are defined as follows.

$$T^* ::= \perp \mid \ulcorner n \urcorner \mid \mathsf{error}_1 \mid \mathsf{error}_2 \mid \mathrm{list}_m(T^*, \ldots)$$

where $m \in \mathbf{N} \geq 1$, or $m = \omega$. Moreover, for each $\mathrm{list}_m(\ldots)$, there are m subtrees. The approximation ordering is defined as

$$\begin{array}{lrcl}
\forall x \in T^* & \perp & \sqsubseteq & x \\
\forall x \in T^* & x & \sqsubseteq & x \\
\forall x_i, y_i \in T^* & \mathrm{list}_m(x_1, x_2, \ldots, x_m) & \sqsubseteq & \mathrm{list}_m(y_1, y_2, \ldots, y_m) \text{ iff } x_i \sqsubseteq y_i, 1 \leq i \leq m.
\end{array}$$

∎

Each $\text{list}_m(\ldots)$ defines a constructor with m open positions. Since m can be ω, infinite branching is permitted. For ease of notation, we introduce a set $\mathbf{N}^* = \mathbf{N} \cup \{\omega\} - \{0\}$.

It is easy to see that the prime basis $(P^*, \leq, \sqsubseteq)$ for the domain $\mathbf{T}^*$ is given as follows.

1. The set of primes P^* is given by the inductive definition:

   $$P^* = \mathsf{error}_1 \mid \mathsf{error}_2 \mid \ulcorner n \urcorner \mid \text{list}_m(\bot, \ldots,) \mid \text{list}_m(\bot, \ldots, p, \bot, \ldots) \qquad p \in P, m \in \mathbf{N}^*$$

   That is, at most one of the positions in $\text{list}_m()$ is not a $\bot$.

2. The ordering relation $\leq$ on P^* is defined by the rules:

   $$\begin{array}{rcll} x & \leq & x & \text{for} \quad x \in \mathbf{N} \cup \{\mathsf{error}_1, \mathsf{error}_2\} \\ \text{list}_m(x_1, \ldots, x_n) & \leq & \text{list}_m(y_1, \ldots, y_n) & \text{iff} \quad [(x_i = \bot) \vee (x_i \leq y_i)] \end{array}$$

3. The *conflict* sets (C-sets) are defined by the equivalence closure of the relation C' over P:

   $$\begin{array}{rcll} (0, \mathsf{error}_i) & \in & C' & \\ (0, n) & \in & C' & \text{for} \quad n \in \mathbf{N} \\ (0, \text{list}_m(\bot, \ldots)) & \in & C' & m \in \mathbf{N}^* \\ (\text{list}_m(x_1, \ldots, x_m), \text{list}_m(y_1, \ldots, y_m)) & \in & C' & \text{if} \quad \exists i : [(x_i, y_i) \in C' \\ & & & \wedge \forall j \neq i : x_j = y_j] \end{array}$$

4. The two error elements in a C-set are the elements that have error_1 and error_2 at the end of the path.

For example, the elements $\text{list}_1(\bot)$ and $\text{list}_2(\bot, \bot)$ are in the same C-set. The error elements in this C-set are error_1 and error_2. The elements $\text{list}_1(2)$ and $\text{list}_1(3)$ are in the same C-set. The corresponding error elements are $\text{list}_1(\mathsf{error}_1)$ and $\text{list}_1(\mathsf{error}_2)$.

The domain $\mathbf{T}^*$ has the required elements that help us preserve finiteness of elements in the embedding. As with the previous embedding, only the non-error elements of the domain are presented. The embedding function provides two error elements for each C-set. The embedding process is similar to that of in the domain $\mathbf{T}$. In particular, the domain $\mathbf{T}$ can be embedded into $\mathbf{T}^*$ and vice versa.

Induction Hypothesis: The function f maps the prime elements into $\mathbf{T}^*$ such that the following properties are satisfied.

1. If $(p, q) \in C$, then $f(p) \# f(q)$ in $\mathbf{T}^*$. That is, conflicting primes are mapped to inconsistent finite elements.
2. If $p, q \in a$ where a is a finite element of D, that is, p, q can form a finite-prime ideal, then $f(p) \uparrow f(q)$.
3. If $p \leq q$, then $f(p) \sqsubseteq f(q)$.
4. For every p_i, $f(p_i)$ contains a specially marked sublist s_i such that s_i has exactly the same number of $\perp$'s as the number of C-sets above p_i.
5. Every prime element is mapped to a finite element in $\mathbf{T}^*$.

Base Case: f maps p_0, which must be $\perp$, to $\text{list}_l(\perp, \ldots)$ where l is the number of C-sets above $\perp$. The difference between l and **arity** is that **arity** must represent ω in $\mathbf{N}$, where as l could be ω. If the **arity** is ω, the subtree is $\text{list}_\omega(\perp, \perp, \perp, \ldots)$. That means, there are exactly as many number of $\perp$'s as there are C-sets above p_0. The subtree s_0 is the entire tree. Thus, the induction hypothesis can be verified.

Induction Step: Assume that we have embedded all the primes in the enumeration from p_0 through p_{i-1}. We define $f(p_i)$ as follows:

Let k be the cardinality of the set of the already enumerated elements that are in conflict with p_i. That is, k prime elements belonging to the C-set of p_i have been enumerated so far. Define s_i, a subtree for p_i as $\text{list}_{m+1}(k, \perp^m)$ where m is the number of C-sets above p_i.

We insert the subtree in the whole tree as follows. Let q be the element immediately below p_i. By induction, q has been embedded in $\mathbf{T}^*$ as a tree $f(q)$. The element q was embedded in $\mathbf{T}^*$ by adding a subtree s_q to the image of q's predecessor. In the subtree s_q, insert s_i (the subtree corresponding to p_i) at position $g(p_i)$, where the function $g(p_i)$ is defined as follows. If a prime p' belonging to the same C-set as p_i has been enumerated, then $g(p_i) = g(p')$. Otherwise, $g(p_i)$ is the number of distinct C-sets above q that are already enumerated. By the induction hypothesis, the position $g(p_i)$ has $\perp$ in the subtree s_q.

Verification of induction hypothesis can be adapted from the previous embedding in $\mathbf{T}$.

Now, we have a mapping from prime elements, including $\perp$, to the finite elements of the tree domain. The finite elements of D can be mapped as the least upper bounds of the mappings of their prime approximations. We can show the following important property of the embeddings.

Proposition 5.12 The preceding mapping maps a finite element of D to a finite element of $\mathbf{T}^*$. It also maps a maximal element of D to a maximal element of $\mathbf{T}^*$.

Proof. Since a finite element is a lub of a finite set of prime elements, and each prime element is mapped to finite element, the finite element is mapped to a finite element in $\mathbf{T}^*$.

Consider a finite maximal element $d \in D$. The elements on the fringe must have **arity** as 0. That is, there are no C-sets above the prime elements on the fringe. Hence, the corresponding subtrees in the embeddings do not have $\perp$'s.

Any prime element p not in the fringe must have $\perp$'s in its subtree. Since d is maximal, d must contain one prime from each C-set above p. Thus, all the $\perp$'s in the subtree for p are filled in the mapping for a. Therefore, the mapping of d does not contain any $\perp$'s, hence, a maximal tree. $\square$

The proof that the preceding image can be embedded by a partial closure is similar to that of the previous embedding into $\mathbf{T}^*$. Therefore, given a domain D with an effective presentation, we can construct a partial closure whose fixed point set if isomorphic to D. In addition, if an element in D is finite, its image in $\mathbf{T}^*$ is finite maximal as well.

Example 5.7 Consider the domain $\mathbf{N}_{\perp}^{E}$. As per the preceding construction, we embed $\perp$ as $\mathrm{list}_1(\perp)$. If the enumeration assigns n to each number n, the subtree for each natural number is $\mathrm{list}_1(n)$. Hence, the subdomain consists of $\mathrm{list}_1(\perp)$ and $\mathrm{list}_1(\mathrm{list}_1(n))$, $n \in \mathbf{N}$. The function that embeds can be written as $\{(\perp, \mathrm{list}_1(\perp)), (\mathsf{error}_i, \mathrm{list}_1(\mathsf{error}_i)), (n, \mathrm{list}_1(\mathrm{list}_1(n))) \mid n \in \mathbf{N}\}$. ∎

Example 5.8 Consider the domain T^ω. Its embedding into $\mathbf{T}$ does not preserve finiteness of elements. However, we can embed it into $\mathbf{T}^*$ preserving the finiteness property. Consider the embedding of $\langle \perp, \ldots \rangle$. It is mapped to $\mathrm{list}_\omega(\perp, \ldots)$ which is a finite element in $\mathbf{T}^*$. Similarly, we can show that any finite element in the domain T^ω is mapped to a finite element in $\mathbf{T}^*$. ∎

Clearly, $\mathbf{T}^*$ preserves all the properties that $\mathbf{T}$ preserves. In addition, it maps finite elements to finite elements. However, this benefit comes with an additional complication of the domain. The domain $\mathbf{T}^*$ can be thought of as a quotient of the domain $\mathbf{T}$. The element $\mathrm{list}_m(\ldots)$ is a synonym for $\langle m, \langle ., . \rangle \rangle$. Thus, we can embed $\mathbf{T}^*$ into $\mathbf{T}$. For rest of our analysis, we use $\mathbf{T}$ as it preserves the essential aspect of computation, namely, the finite maximality of answers.

Chapter 6

Universal Language

In the preceding chapter, we have introduced the domain $\mathbf{T}$ and showed that it is a universal domain. In particular, any OS-domain D can be embedded in $\mathbf{T}$ by an OS-function. This embedding provides a mechanism to provide semantics of operations D. However, to perform any computations, we require a language to compute.

In this chapter, we present a universal language KL, short for Kleene's Language. It is a first order language with recursion. It comes with a set of primitives on $\mathbf{T}$, and *Apply*. We provide the denotations for invocation of programs in KL. We also give a reduction semantics for expressions in KL and show that it matches the denotational semantics. We end the chapter by showing that KL can express any computable function over or computable element of $\mathbf{T}$.

6.1 The Language KL

In this section, we introduce the programming language KL for the universal domain $\mathbf{T}$. This language is based on first order recursion equations. A program, consisting of a set of recursion equations, introduces new function symbols, which an invocation uses to denote an element of $\mathbf{T}$. We first present a general recursive programming scheme and later introduce the language KL.

The recursive program scheme comes with a set of primitive functions to operate on the data domain. Let $G = \{g_1, \ldots, g_n\}$ be the set of primitive functions on the given domain $\mathbf{T}$.[15] In addition to G, the recursion equations define a set of function symbols. We can define an expression from these function symbols with free variables as follows.

Definition 6.1. (*Expression*) An *expression* over (G, F, V) where G is the finite set of primitives $\{g_i\}$, F is the finite set of operations $\{f_i\}$, and V is the finite set of variables $\{x_i\}$ is:

1. a constant symbol $c \in F \cup G$, *i.e.*, a symbol that takes no arguments,
2. a variable $v \in V$, or
3. an application $g(\alpha_1, \ldots, \alpha_n)$ where g is an n-ary function symbol in $F \cup G$ and $\alpha_1, \ldots, \alpha_n$ are simpler expressions over (G, F, V). These simpler expressions are called *subexpressions*.

∎

It is evident from the definition of expression that if a function symbol appears in the expression, it appears in the application position with all the arguments it can take. This is in contrast to the languages with higher-order functions, where functions can be passed as arguments and returned as arguments. Higher order functions are treated as a tree in KL; since the tree representation is extensional, we do not lose order-extensionality of functions in the model.

To interpret an expression over (G, F, V), we introduce the concept of *environment*.

Definition 6.2. (*Environment*) An environment σ is a function mapping $V \cup F$ into $\mathbf{T}$. ∎

For the expressions over (G, F, V) the meaning function $\mathbf{E} : expr \rightarrow (env \rightarrow \mathbf{T})$ is defined inductively in the obvious way.

[15]Strictly speaking, we have to distinguish between the syntactic representation of a function in G and its denotation. Since it is evident from the context whether we are using the function in the syntax or the semantics, we do not use different notation.

1. $\mathbf{E}[\![f]\!]\sigma = \sigma(f)$ for constant symbols $f \in F$.

2. $\mathbf{E}[\![v]\!]\sigma = \sigma(v)$ for variables $v \in V$.

3. $\mathbf{E}[\![g]\!]\sigma = g$ for $g \in G$.

4. $\mathbf{E}[\![h(\alpha_1, \ldots, \alpha_n)]\!]\sigma = \sigma(h)(\mathbf{E}[\![\alpha_1]\!]\sigma, \ldots, \mathbf{E}[\![\alpha_n]\!]\sigma)$.

The meaning of an expression in (G, F, V) is a mapping from an environment to an element in the domain.

A recursive program P is a set of functions

$$f_1(\vec{x}_1) = \tau_1, \ldots, f_n(\vec{x}_n) = \tau_n$$

where the f_i are the symbols distinct from G and τ_i is an expression over $(G, F, \vec{x}_i)$. If the variable list is empty then the function is a constant.

A recursive program defines a set of function symbols that are used in the invocation of the program. Semantically, a program defines an environment which assigns meanings to the function symbols. An invocation of a program is evaluated in that environment.

Definition 6.3. (*Invocation of a Program*) Let F be the set of function symbols introduced in the program P. An invocation of the program P is an expression over $(G, F, \emptyset)$. ∎

An invocation is an expression without any free variables. Thus, it is an expression composed of the function symbols and the primitive operations. Its meaning can be given by environment defined by the program.

6.1.1 KL: The Primitive Operations

The language KL is based on recursive program scheme. It includes a binary operator *Apply*: $\mathbf{T} \times \mathbf{T} \rightarrow_{os} \mathbf{T}$ and a set of constants: $\{0, \mathsf{error}_i, \mathbf{1^+}, \mathbf{1^-}, \mathbf{pair?}, \mathbf{left},$

right,**cons**, **if0**}. These primitives construct trees, select subtrees, and perform arithmetic operations on natural numbers. All the constants except 0 and error_i are elements of $\mathbf{T} \to_{os} \mathbf{T}$ (embedded in $\mathbf{T}$); they are interpreted as functions by *Apply*. For notational convenience we use them as bona fide uncurried functions, without showing the implicit *Apply*.[16]

Example 6.1 A program written in KL consists of a set of recursive definitions and an invocation of the program. For example, the set of recursive definitions can be empty and the invocation may use just the primitive operations:

$$\mathbf{cons}(1, 2)$$

Or, the recursion equations can define the function symbols to be used in the invocation:

$$\begin{aligned} fact(x) &= \mathbf{if0}(x, 1, mult(x, fact(\mathbf{1}^{-}(x)))) \\ mult(x, y) &= \mathbf{if0}(x, x, add(x, mult(\mathbf{1}^{-}(x), y))) \\ add(x, y) &= \mathbf{if0}(x, y, add(\mathbf{1}^{-}(x), \mathbf{1}^{+}(y))) \end{aligned}$$

The invocation of the program can be:

$$fact(10)$$

$$fact(mult(1, \mathbf{1}^{+}(1)))$$

The recursion equations define the function symbols $fact, mult, add$. The invocations of the program use the function $fact$, $mult$ in conjunction with the primitive operation $\mathbf{1}^{+}$. ■

6.1.2 Denotational Semantics

The denotational semantics for a program is primarily concerned with providing a meaning for an invocation of the program. To that end, it assigns meaning to all the

[16] Technically, all the basic operations are trees; and *Apply* treats these trees as functions appropriately. These trees are dependent on the embedding function we choose.

primitives and the function symbols. An expression e, which is the invocation of a program P is assigned a meaning as follows.

1. Each primitive operation is assigned a meaning.
2. The program P defines an environment σ_p, which maps all the function symbols to their meanings.
3. Since the expression is a mapping from an environment to an element, we evaluate it in the environment defined by the program.

Denotations for the Primitives

First, we provide the denotations for each of the primitive functions. As stated earlier, all the primitive operations are trees; they are interpreted as functions by *Apply*. Since we are more interested in using these primitive operations as functions, we provide the functional representation of the tree. In these representations n represents a natural numbers in $\mathbf{T}$ and x, y, z represent elements of $\mathbf{T}$.

All the above primitive functions are OS-functions, hence can be embedded into $\mathbf{T}$ such that *Apply* can interpret them appropriately. Since these functions are trees, we can pass them as parameters. That is, $Apply(\mathbf{left}, \mathbf{right})$ is a valid operation, as **right** is a constant tree.

Lemma 6.1 For any expression e over $(G, \emptyset, \emptyset)$, $\mathbf{E}[\![e]\!]\sigma \in \mathbf{T}$.

Proof. We can prove it by structural induction on the expressions. The environment σ does not play any role in assigning the denotation. □

Denotation of a Program

To define the meaning of an invocation of a program, we must define the meaning of the function symbols. The meaning of an n-ary function f is defined under the

The constants:

$$\begin{aligned}
\mathbf{E}[\![0]\!]\sigma &= 0\\
\mathbf{E}[\![\mathsf{error}_i]\!]\sigma &= \mathsf{error}_i
\end{aligned}$$

Unary Operations

$$\begin{aligned}
\mathbf{E}[\![\mathbf{1^+}]\!]\sigma &= \{n \to n+1\} \cup \{\mathsf{error}_i \to \mathsf{error}_i\}\\
\mathbf{E}[\![\mathbf{1^-}]\!]\sigma &= \{0 \to 0\} \cup \{n+1 \to n\} \cup \{\mathsf{error}_i \to \mathsf{error}_i\}\\
\mathbf{E}[\![\mathbf{pair?}]\!]\sigma &= \{\langle \perp, \perp \rangle \to 0\} \cup \{n \to 1\} \cup \{\mathsf{error}_i \to \mathsf{error}_i\}\\
\mathbf{E}[\![\mathbf{left}]\!]\sigma &= \{\langle x, \perp \rangle \to x\} \cup \{\mathsf{error}_i \to \mathsf{error}_i\}\\
\mathbf{E}[\![\mathbf{right}]\!]\sigma &= \{\langle \perp, y \rangle \to y\} \cup \{\mathsf{error}_i \to \mathsf{error}_i\}
\end{aligned}$$

Binary Operations

$$\mathbf{E}[\![\mathbf{cons}]\!]\sigma = \{x \times y \to \langle x, y \rangle\}$$

Ternary Operations

$$\mathbf{E}[\![\mathbf{if0}]\!]\sigma = \{0 \times x \times \perp \to x\} \cup \{1 \times \perp \times y \to y\} \cup \{\mathsf{error}_i \times \perp \times \perp \to \mathsf{error}_i\}$$

Figure 6.1 Denotations of the Primitive Functions in KL

environment σ as:

$$\mathbf{E}[\![f]\!]\sigma = \{\vec{a}_i \to \mathbf{E}[\![\tau_i]\!]\sigma[\vec{x}_i/\vec{a}_i]\}.$$

In the case where the function has no parameters, it reduces to

$$\mathbf{E}[\![f_i]\!]\sigma = \mathbf{E}[\![\tau_i]\!]\sigma.$$

Thus, an n-ary function is assigned an n-ary function in $\mathbf{T}$ as its meaning.

However, if the function is defined recursively, to provide the meaning for f, f itself must be defined in σ. In the following definitions, we will unroll the definition of f into f_i's that do not contain f. Thus, the meaning of f is dependent on the meanings of f_i's. Eventually, we will show that we can construct an environment where $\mathbf{E}[\![f]\!]\sigma = \sigma(f)$, *i.e.*, the meaning is a fixed point for the equation.

As stated earlier, a program defines an environment where all the function symbols are mapped to their meanings. We define that environment σ as a limit of σ_i's defined as follows.

- $\sigma_0 = \bot$, *i.e.*, $\sigma(f_i) = \bot$.

- $\sigma_n = \sigma_{n-1}[f_i/\mathbf{E}[\![f_i]\!]\sigma_{n-1}]$. That is, $\sigma_n(f_i) = \mathbf{E}[\![f_i]\!]\sigma_{n-1}$.

Each σ_i defines a meaning of the function f, where the function applications are expanded to depth i. To define σ_p as the limit of σ_i we must show that σ_i's form a chain.

Since each σ_i is a mapping from symbols into $\mathbf{T}$ or functions over $\mathbf{T}$, σ_i can be treated as functions over $\mathbf{T}$. That is, $\sigma \sqsubseteq \sigma'$ iff for any symbol s, $\sigma(s) \sqsubseteq \sigma'(s)$. We prove some useful lemmas about environments which we use later in our proofs.

Lemma 6.2 If $\sigma \sqsubseteq \sigma'$ then $\mathbf{E}[\![\tau]\!]\sigma \sqsubseteq \mathbf{E}[\![\tau]\!]\sigma'$.

Proof. We prove the lemma by structural induction on the terms.

Base Case: For the constants, variables, and functions it is trivially true.

Induction Step: Consider an expression $f(\vec{a})$. $\mathbf{E}[\![f(\vec{a})]\!]\sigma = \sigma(f)(\mathbf{E}[\![\vec{a}]\!]\sigma)$. By IH, for each a in the arguments, $\mathbf{E}[\![a]\!]\sigma \sqsubseteq \mathbf{E}[\![a]\!]\sigma'$. Also, $\sigma(f) \sqsubseteq \sigma'(f)$. Therefore, by monotonicity of OS-functions, we have the required result.

$\square$

Lemma 6.3 The set of σ_i form a chain in the domain $\mathbf{T}$.

Proof. First, we must prove that each σ_i maps symbols into functions over or elements of $\mathbf{T}$. Next, we must show that they form a chain. We prove it by induction on i.

Induction Hypothesis: The environments σ_i form a chain in $\mathbf{T}$.

Base Case: σ_0 is an OS-function on $\mathbf{T}$, mapping function symbols to OS-functions over $\mathbf{T}$, namely the $\bot$ function. Since it is the only element, it obviously forms a chain of one element. Also, σ_0 maps any function symbol into $\bot$, which is an element of $\mathbf{T}$.

Induction Step: Let σ_i be a chain in $\mathbf{T}$. For a function f, $\sigma_{i+1}(f)$ is $\mathbf{E}[\![f]\!]\sigma_i$. That is, it a function defined as $\{\vec{a} \to \mathbf{E}[\![\tau]\!]\sigma_i[\vec{x}/\vec{a}]\}$. Since $\sigma_{i-1} \sqsubseteq \sigma_i$ by IH, this function must dominate $\{\vec{a} \to \mathbf{E}[\![\tau]\!]\sigma_{i-1}[\vec{x}/\vec{a}]\}$ by 6.2. Therefore, $\sigma_i \sqsubseteq \sigma_{i+1}$. Also, $\sigma_{i+1}(f)$ is equal to $\{\vec{a} \to \mathbf{E}[\![\tau]\!]\sigma_i[\vec{x}/\vec{a}]\}$. By IH, σ_i maps the functions to OS-functions. Hence $\mathbf{E}[\![\tau]\!]\sigma_i[\vec{x}/\vec{a}]$ is in $\mathbf{T}$.

Using structural induction of τ, we can show that $\{\vec{a} \to \mathbf{E}[\![\tau]\!]\sigma_i[\vec{x}/\vec{a}]\}$ is an OS-function. The proof hinges on the fact that the composition of OS-functions is an OS-function.

Therefore, σ_i's form a chain in $\mathbf{T}$. □

The meaning of a given program P is captured in the environment $\sigma_p = \bigsqcup \sigma_i$. Any function symbol f in P is assigned a meaning $\sigma_p(f)$. This meaning is the least fixed point of the equation $f(\vec{x}) = \tau$.

Lemma 6.4 If the program P is $f(\vec{x}) = \tau$ then the meaning of f is the least fixed point of the equation.

Proof. f is the least fixed point of the equation iff f is the least fixed point of the functional $F = \lambda f.\lambda\vec{x}.\tau$. Since the meaning of f is $\sigma(f)$, we must show that $F(\sigma(f)) = \sigma(f)$. Since $F(\sigma(f)) = \{\vec{a} \to \mathbf{E}[\![\tau]\!]\sigma\}$, we must show that the right hand side is $\sigma(f)$.

$$\begin{aligned}
\{\vec{a} \to \mathbf{E}[\![\tau]\!]\sigma\} &= \{\vec{a} \to \textstyle\bigsqcup \mathbf{E}[\![\tau]\!]\sigma_i\} \\
&= \textstyle\bigsqcup\{\vec{a} \to \mathbf{E}[\![\tau]\!]\sigma_i\} \\
&= \textstyle\bigsqcup \sigma_{i-1}(f) \\
&= \sigma(f)
\end{aligned}$$

Therefore, $\sigma(f)$ is a fixed point of f.

To show that it is a least fixed point, observe that $F(\sigma_i(f)) = \sigma_{i+1}(f)$, hence for any fixed point of F, we have the following reasoning. If $F(g) = g$, since $g \sqsupseteq \sigma_0(f)$, by applying F on both sides, we have $g \sqsupseteq \sigma_1(f)$. Thus, inductively, $g \sqsupseteq \sigma_i(f)$, and therefore $g \sqsupseteq \sigma(f)$. Thus, $\sigma(f)$ is the least fixed point. □

If the program consists of several recursion equations $f_i(\vec{x_i}) = \tau_i$, then it can easily be shown that $F = [f_1, f_2, f_3 \ldots, f_n]$ is the least fixed point of the functional $\lambda F.(\lambda f_1, \ldots f_n.[\lambda \vec{x_1}.\tau_1, \ldots, \lambda \vec{x_n}.\tau_n])(\pi_1(F), \ldots, \pi_n(F))$. Here, $[\ldots]$ is used to denote n-ary product, and π_i projects the i'th component.

The meaning of an invocation e of a program P is defined as $\mathbf{E}[\![e]\!]\sigma_p$ where σ_p is the environment defined by the program P. Since the meaning of e a composition of OS-functions over elements of $\mathbf{T}$, it is a tree in $\mathbf{T}$.

6.1.3 Operational Semantics

Operational semantics provides an operational model of the execution of a program. Operation semantics specify a systematic *reduction* of a program until no further reduction is possible. At that stage, the execution is said to reach an answer.

Operational semantics is specified by a set of rewriting rules. The rewriting rules consist of two mutually independent sets:

δ-rules These rules define the operational semantics for the primitive functions. They are listed in Fig. 6.2. As usual, we use $f(x)$ when we mean $Apply(f, x)$.

β-rule This rule helps us to expand the function invocation. Let F be the set of function symbols introduced in P. If a function is given by the equation $f(\vec{x}) = \tau$ where τ is an expression over $(G, F, \vec{x})$, the β-rule states that:

$$f(\vec{a}) \longrightarrow \tau[\vec{x}/\vec{a}].$$

Notice that δ-rules are not total. That is, when rewriting an expression it is possible to encounter a "stuck" state. For example, $\mathbf{left}(0)$ is a stuck state, since no further rewriting of that term is possible. However, for even a term like 0, which is a final answer, no further rewriting is possible. To differentiate between stuck states and final answers, we introduce a new constant Ω, and extend the δ-rules to include

$$
\begin{array}{rclrcl}
\mathbf{1^+}(n) & \longrightarrow & n+1 & \mathbf{cons}(a)(b) & \longrightarrow & \mathsf{Cons}(a,b) \\
\mathbf{1^+}(\mathsf{error}_i) & \longrightarrow & \mathsf{error}_i & \mathsf{if0}(0)(a)(b) & \longrightarrow & a \\
\mathbf{1^-}(0) & \longrightarrow & 0 & \mathsf{if0}(n+1)(a)(b) & \longrightarrow & b \\
\mathbf{1^-}(n+1) & \longrightarrow & n & \mathsf{if0}(\mathsf{error}_i)(a)(b) & \longrightarrow & \mathsf{error}_i \\
\mathbf{1^-}(\mathsf{error}_i) & \longrightarrow & \mathsf{error}_i & \mathbf{pair?}(\mathsf{error}_i) & \longrightarrow & \mathsf{error}_i \\
\mathbf{pair?}(\mathsf{Cons}(a,b)) & \longrightarrow & 0 & \mathbf{pair?}(n) & \longrightarrow & 1 \\
\mathbf{left}(\mathsf{Cons}(a,b)) & \longrightarrow & a & \mathbf{right}(\mathsf{Cons}(a,b)) & \longrightarrow & b \\
\mathbf{left}(\mathsf{error}_i) & \longrightarrow & \mathsf{error}_i & \mathbf{right}(\mathsf{error}_i) & \longrightarrow & \mathsf{error}_i
\end{array}
$$

Figure 6.2 δ-rules for KL

the following.

$$
\begin{array}{rcll}
g(\Omega) & \longrightarrow & \Omega & \text{where } g \in \{\mathbf{1^+}, \mathbf{1^-}, \mathbf{pair?}, \mathbf{left}, \mathbf{right}\} \\
\mathsf{if0}(\Omega)(a)(b) & \longrightarrow & \Omega & \\
g(\mathsf{Cons}(a,b) & \longrightarrow & \Omega & \text{where } g \in \{\mathbf{1^+}, \mathbf{1^-}\} \\
g(\ulcorner n \urcorner) & \longrightarrow & \Omega & \text{where } g \in \{\mathbf{left}, \mathbf{right}\}
\end{array}
$$

The rewriting rules define a schema of rewriting. We can derive an infinite number of rewriting rules by properly instantiating the variables n with the natural numbers and a, b, c with expressions over any (G, F, V). Unlike the original δ-rules, these extended rules are "total". That is, if a function symbol appears in the expression, it expression can always be rewritten using one of the rewriting rules.

Given an expression over (G', F, V) where $G' = G \cup \{\Omega\}$, we can define the reduction semantics for a set of rewriting rules as follows.

Definition 6.4. (*Reduction*) An expression e reduces to e' iff

- $e \longrightarrow e'$ by an instantiation of one of the rewriting rules, or
- e' is the result of replacing a subexpression e_1 in e with another subexpression e_2 where $e_1 \longrightarrow e_2$.

The transitive closure of $\longrightarrow$ is written as $\longrightarrow^*$. ∎

Reduction of a term may end in an answer, *i.e.*, a term which cannot be reduced further. These terms are called *normal forms*. For the expressions over $(G', \emptyset, \emptyset)$, the reductions may end in a normal form belonging to the following set $Norm$:

$$Norm = \{n, \mathsf{error}_1, \mathsf{error}_2, \Omega, \mathsf{Cons}(a, b) \mid a, b \in Norm\}$$

Lemma 6.5 For all the expressions over $(G', \emptyset, \emptyset)$, the normal forms are precisely $Norm$.

Proof. It can easily be verified that all the terms in $Norm$ are indeed normal forms. For the proof in the other direction, assume that e the smallest term that is in normal form, outside $Norm$. Since e does not belong to $Norm$, it must be of the form $\mathsf{Cons}(a, b)$ where a or b do not belong to $Norm$. Since e cannot be reduced further, a and b also must be in normal form. Hence there is a smaller expression than e that is in normal form outside $Norm$ providing the contradiction. Hence, $Norm$ is the complete set of normal forms. □

For the rewriting process to make sense, we must show that the reduction of a term always produces the same normal form. This property is known as Church-Rosser property. In addition, we also need to know if each term in the language has a normal form.

Definition 6.5. (*CR,SN*) A reduction semantics $\longrightarrow$ is Church-Rosser (CR) iff

$$\forall t : t \longrightarrow^* u \wedge t \longrightarrow^* v \Rightarrow (\exists w : u \longrightarrow^* w \wedge v \longrightarrow^* w).$$

In addition, it is strongly normalizing (SN) iff any reduction of a term always terminates with a normal form. ∎

If the reduction semantics has CR and SN properties, then any rewriting process of an expression always terminates and produces the same normal form. The reduction semantics defined by δ-rules possess these properties.

Lemma 6.6 The reduction semantics defined over the expressions in $(G', \emptyset, \emptyset)$ by δ-rules have CR and SN properties.

Proof. Instead of showing the property CR, we show that a term has at most one normal form. Since we can show the property SN also, each term must terminate in a normal form. Hence, CR and SN properties hold for the δ reduction semantics. The proof proceeds with induction on the structure of expression.

Base Case: All the constants such as $n, \mathsf{error}_i, \Omega$ are normal forms, satisfying the IH.

Induction Step: If a, b, c are expressions in $(G', \emptyset, \emptyset)$, then we will show that any expression formed from them always rewrites to a normal form.

Take the expression $e = \mathbf{1}^+(a)$. It can be rewritten iff a is a number, an error element, Ω, or a cons node with two expressions. Let the normal form of a be a'. We can do the following case analysis on a'.

Case 1. $a' \in \{n, \mathsf{error}_i, \Omega\}$: It is clear that $\mathbf{1}^+(a) \longrightarrow^* \mathbf{1}^+(a')$ which produces a normal form. We cannot rewrite e, until we rewrite a into one of $\{n, \mathsf{error}_i, \Omega, \mathsf{Cons}(a_1, a_2)\}$ where a_1 and a_2 need not be normal forms. By IH, a can never rewrite to a cons node, at any intermediate stage. Also, the rewriting always terminates producing the same normal form. Hence, we always get the same normal form for e in whichever way we rewrite a.

Case 2. $a' = \mathsf{Cons}(a'_1, a'_2)$: By IH, the rewriting of a always terminates in the same normal form. Hence, a can never produce a term in $\{n, \mathsf{error}_i, \Omega\}$. However, in the rewriting process a can yield $\mathsf{Cons}(a_1, a_2)$, where a_1 and a_2 are not in normal forms, which makes $\mathbf{1}^+(a)$ rewrite to the normal form Ω. Even if a_1 and a_2 are in normal forms, $\mathbf{1}^+(a)$ produces the same normal form. Thus, e cannot be reduced until a is reduced up to a cons node and any reduction of a must produce a cons node. Hence, the reduction of e always terminates in Ω.

The rest of the cases of the structural induction are similar.

□

So far, we have dealt with the expression over $(G', \emptyset, \emptyset)$. Since the invocation of a program may contain function symbols introduced by the program, we must provide the rewriting rules for the function applications also.

For each function $f_i(\vec{x_i}) = \tau_i$, we inductively define a sequence of f_i^j's as follows:

- $f_i^0(\vec{x_i}) = \Omega$.
- $f_i^k(\vec{x_i}) = \tau_i[F/F^{k-1}]$, where F is the set of function symbols and F^{k-1} are the set of function symbols $f_1^{k-1} \ldots, f_n^{k-1}$. $\tau[F/F^k]$ is the expression obtained by replacing each function symbol f with f^k.

It is clearly evident that f^k expands the function application up to depth k.

Since an invocation of a program produces increasingly better approximations to the result, we define the evaluation by a limiting process. To that end, we define a series of expressions e_j for a given expression e as $e[F/F^j]$. We define a series of evaluations $eval_i$ of e as:

$$eval_i(e) = e'_i \text{ where } e_i \longrightarrow^* e'_i \text{ and } e'_i \in Norm.$$

Here we extend the reduction process to include β-rule.

Lemma 6.7 Any expression e_k over $(G', \bigcup_{j \leq k} F^j, \emptyset)$ is strongly normalizable. In addition, it has a unique normal form.

Proof. We prove the lemma by induction on k.

Base Case: e_0 clearly is an expression over $(G', \emptyset, \emptyset)$. By lemma 6.6, we know that any rewriting of e_0 terminates with the same normal form.

Induction Step: Let e_{k+1} be an expression containing f^{k+1}'s. Let ρ be a reduction process on e that does not terminate. If this reduction process does not reduce any of the functions f^{k+1}'s then we can produce a non-terminating reduction sequence of an expression over $(G', \bigcup_{j \leq k} F^j, \emptyset)$ obtained by replacing all the function f^{k+1} applications to Ω. However, by IH, this expression always terminates in a normal form, hence provides a contradiction.

Let ρ_1, ρ_2 be two reduction sequences on e_{k+1} ending in two different normal forms. Let e be e_{k+1} with f^{k+1} applications expanded out so that it contains only f^k's. We can derive ρ_1' and ρ_2' on e by removing the reduction steps that expand f^{k+1}'s into f^k's from ρ_1 and ρ_2 . If ρ_1 and ρ_2 produce different results on e_{k+1}, ρ_1', ρ_2' produce different normal forms for e, which contradicts IH.

□

Relating normal forms to trees

Since normal forms are assumed to be answers, we must relate them to the trees. We map normal forms to trees as follows. Using this mapping, we can relate the denotations of bodies to their normal forms. The function $\mathcal{D}$ is a bijective mapping from normal forms to elements of $\mathbf{T}$.

$$\begin{array}{rcl} \mathcal{D}(n) & = & n \\ \mathcal{D}(\mathsf{error}_i) & = & \mathsf{error}_i \\ \mathcal{D}(\Omega) & = & \bot \\ \mathcal{D}(\mathsf{Cons}(a,b)) & = & \langle \mathcal{D}(a), \mathcal{D}(b) \rangle \end{array}$$

It is easy to see that the function $\mathcal{D}$ is one-to-one and onto. We can easily construct the normal form denoting a given tree.

To show that the denotational semantics matches with the operational semantics we must show that the denotation of a term relates to its normal form. In other words, the denotation of the term is identical to the tree that is derived from mapping its normal form. We prove this result using the following lemmas.

Lemma 6.8 *Adequacy:* Let e be an expression over (G, F, V). Then for $\vec{v} \subseteq V$, and $\vec{a} \subset Norm$, $\mathbf{E}[\![e]\!]\sigma_i[\vec{v}/\mathcal{D}(\vec{a})] = \mathcal{D}(eval_i(e[\vec{v}/\vec{a}])$ for all i.

Proof. We prove the lemma by induction on i.

Base Case: For σ_0, we proceed with structural induction on e.

Base Case: For 0, error_i, or Ω, the equality clearly holds. For any variable $v \in V$ also it holds trivially.

Induction Step: For all the primitive functions, the equality holds by simple check on the δ-rules. For a function application $f(\vec{\alpha})$, since the application of $\sigma_0(f)$ to any arguments is $\perp$, and $eval_0(f(\vec{a}))$ is Ω, the equality holds.

Induction Step: Let the equality hold for σ_i until i. We prove the equality for $i+1$, by structural induction of e.

Base Case: For 0, error_i, and Ω, the equality holds. For all the variables in V the equality holds.

Induction Step: For all the primitive functions and variables in V, the equality holds. For a function application $f(\vec{\alpha})$, we can show the equality as follows.

Let σ be $\sigma_{i+1}[\vec{v}/\vec{\mathcal{D}(a)}]$ and σ' be $\sigma_i[\vec{v}/\vec{\mathcal{D}(a)}]$. Let $\vec{b}$ be the denotation of α under σ. Let $\vec{c} = eval_i(\vec{\alpha})$. By IH, we know that $\vec{b} = \mathcal{D}(\vec{c})$.

$$\begin{array}{rcll} \mathbf{E}[\![f(\vec{\alpha})]\!]\sigma & = & \mathbf{E}[\![\tau]\!]\sigma'[\vec{x}/\vec{b}] & \text{Since } \mathbf{E}[\![\vec{\alpha}]\!]\sigma = \vec{b} \\ & = & \mathcal{D}(eval_i(\tau[\vec{x}/\vec{c}][\vec{v}/\vec{a}])) & \text{by IH} \\ & = & \mathcal{D}(eval_{i+1}(f(\vec{c})[\vec{v}/\vec{a}])) & \text{by definition of } eval_{i+1} \end{array}$$

□

Corollary 6.6 For an expression over $(G, F, \emptyset))$, for all i, $\mathcal{D}(eval_i(e)) \sqsubseteq \mathcal{D}(eval_{i+1}(e))$.

Proof. By lemma 6.8, we know that $\mathcal{D}(eval_i(e)) = \mathbf{E}[\![e]\!]\sigma_i$. Therefore, by lemma 6.2, the required approximation holds. □

Adequacy

To show that the rewriting rules are sound with respect to the denotational semantics, we must relate the answers from the execution of the programs to their denotations.

An invocation of a program can have two different kind of evaluation strategies. One that evaluates only on demand, and one that evaluates the expression completely. A lazy evaluator[17] would just print the top level value of the element, *i.e.*, a number or error value or a cons node. If the evaluator prints a cons node, further demands can be made to evaluate the left branch or the right branch. A demand for evaluating the left branch of an expression e can be thought of lazily evaluating **left**(e). Thus, the lazy evaluator produces only the *top level node*. On the other hand, the full evaluator evaluates the tree fully printing all the finite approximations to the answer.

Definition 6.7. (*Top Level Node*) For an expression $e \in \mathit{Norm}$, the top level node is defined as n if $e = n$, error_i if $e = \mathsf{error}_i$ and Cons if $e = \mathsf{Cons}(.,.)$. ∎

Definition 6.8. (*Lazy Evaluation*) Let e be an expression over $(G, F, \emptyset)$. Then $\mathit{LazyEval}(e)$ is the top level node of $\mathit{eval}_i(e)$ where i is the least number for which $\mathit{eval}_i(e)$ is not Ω. ∎

By corollary 6.6, it is clear that if the top level node of $\mathit{eval}_i(e)$ is defined then it is same as the top level node of $\mathit{eval}_{i+1}(e)$. If the evaluation always leads to Ω for any i, then the top level node for any $\mathit{eval}_i(e)$ does not exist. Then the evaluation is said to *diverge*. We can relate the lazy evaluation to the denotation as follows.

Theorem 6.9 Let e be an expression over $(G, F, \emptyset)$. The lazy evaluation of e diverges iff its denotation is $\perp$. The lazy evaluation respectively yields n, or error_i, or a *Cons* node iff the denotation of e is n, or error_i, or $\langle ., . \rangle$ respectively.

Proof. If the lazy evaluation of e diverges, then there is no i for which $\mathit{eval}_i(e)$ is not Ω. By Lemma 6.8, we infer that there is no i for which $\mathbf{E}[\![e]\!]\sigma_i$ is not $\perp$. Since the denotation is the limit of $\mathbf{E}[\![e]\!]\sigma_i$ over i's, we know that the denotation must be $\perp$.

[17] Obviously lazy evaluation is not call-by-name.

If the lazy evaluation produces n, then there must be an i for which $eval_i(e)$ is n. Hence, by Lemma 6.8, $\mathbf{E}[\![e]\!]\sigma_i$ must be n. Therefore, the denotation must dominate n, and since n is maximal, must be equal to n.

If the lazy evaluation produces a cons node, then there must be i, for which $eval_i(e)$ is $\mathsf{Cons}(.,.)$. Therefore, the denotation must dominate $\langle \perp, \perp \rangle$, proving the lemma.

The case of error_i is similar to that of n. □

As opposed to lazy evaluation, which evaluates only the top level node, we can define *full evaluation*. This strategy evaluates the normal form completely, printing out the intermediate answers. That is, the process prints a node as soon as it evaluates to a value other than bottom. Thus, it progressively displays the tree while it evaluates. This process stops only when the whole tree is evaluated.

Definition 6.10. (*Full Evaluation*) Let e be an invocation of a program P defining a set of function symbols F. Then, $\mathit{FullEval}(e)$ is the tree $\mathcal{D}^{-1}(\bigsqcup \mathcal{D}(eval_i(e)))$. ∎

It is obvious that the full evaluation stops only when the whole tree is displayed. Speaking from the domain point of view, only the computations that denote finite maximal elements terminate. If a computation denotes a partial element, then the evaluation process either gets stuck or enters a loop at some node of the tree.

Theorem 6.11 Let e be an invocation of a program P. Then $\mathit{FullEval}(e)$ terminates, *i.e.*, does not contain Ω's iff $\mathbf{E}[\![e]\!]\sigma_p$ is a finite maximal element. Here σ_p is the environment defined by P.

Proof. By Lemma 6.8, $\mathbf{E}[\![e]\!]\sigma_p = \bigsqcup \mathcal{D}(eval_i(e))$, hence it is equal to $\mathcal{D}(\mathit{FullEval}(e))$. Therefore, by the definition of $\mathcal{D}$, the denotation contains a $\perp$ in the tree iff the normal form contains a Ω. If the denotation is a finite maximal element, it cannot contain a $\perp$ at any of its nodes, hence, $\mathit{FullEval}(e)$ cannot contain Ω. If the denotation is not finite, the normal form is not finite, hence the computation cannot terminate. Thus, the computation terminates iff the denotation is a finite maximal element. □

6.2 Computability and Universality

If we want to define an arbitrary OS-domain in KL, we must show that KL can express any computable OS-function. This proof has two parts: First we must show that any partial recursive function over the natural numbers can be defined in KL. Second, we must show that any element or OS-function that is computable on the domain $\mathbf{T}$ can be written in KL.

We show that the language KL is partially recursive by encoding an arbitrary partial recursive function in the language. It is well known [Rog67] that any partial recursive function can be expressed by the operations: $0, \mathbf{1}^+, \mathbf{1}^-$ and selection, composition, primitive recursion, and minimization. All these operations can be expressed in KL over the natural numbers.

We show that the language KL is partially recursive by encoding any given partial recursive function in the language. Since any partial recursive function can be expressed by the following rules [Rog67], we express them in KL in the following way:

1. $0 = \ulcorner 0 \urcorner$.
2. SUC $= \mathbf{1}^+$. PRED $= \mathbf{1}^-$.
3. Zero function is defined by the program $fx = \ulcorner 0 \urcorner$.
4. Selection functions are present in the recursive programming scheme.
5. Composition and recursion is present in this scheme using the other primitives and test for zero. *i.e.*,

$$f(0, \vec{x}) = g(\vec{x})$$

$$f(n+1, \vec{x}) = h(f(n, \vec{x}), n, \vec{x})$$

can be written as

$$f(n, \vec{x}) = \mathbf{if0}(n, g(\vec{x}), h(f((\mathbf{1}^- n), \vec{x}), (\mathbf{1}^- n), \vec{x}))$$

Similarly,

$$k(\vec{x}) = l(g_0(\vec{x}), g_2(\vec{x}), \ldots, g_m(\vec{x}))$$

can be written in the same way.

6. Minimization can be done as in the following: Let $f(\vec{x}) = \mu y.k(y, \vec{x})$ i.e., the value of the function is the minimum y such that $k(y, \vec{x})$ is zero.

   This equation can be written in the RPS:

$$f(\vec{x}) = g(0, \vec{x})$$

$$g(n, \vec{x}) = \mathbf{if0}(k(n, \vec{x}), n, g((\mathbf{1}^{+}n), \vec{x}))$$

A language is universal for a domain, if the language can express all the computable elements and the functions of the domain. In the context of $\mathbf{T}$ the language KL is universal if it can express all the computable elements of $\mathbf{T}$ and the computable subset of OS-functions. Notice that not all OS-functions are computable, hence the language cannot express all the OS-functions.

Given a domain with effective presentation π, an element or an operation is computable if all the approximating finite elements (or, prime elements in the case of sequential domains) are recursively enumerable.

If a function f is computable then $\{\vec{b}, f(\vec{b})\}$ where $\vec{b}$ are finite elements, is recursively enumerable. Since the ordering among the functions is pointwise, this is equivalent to having all the approximating finite elements recursively enumerable.

Given any computable element d, since the set of elements approximating this element are r.e., there is a partial recursive function that has the following property:

$$f(n) = m \iff pos(n, m) \sqsubseteq d$$

where pos(n,m) is the element that can be described in the following functions:

$$
\begin{aligned}
val(n) &= \mathbf{if0}(n, \mathbf{cons}(\perp, \perp), \\
&\quad \mathbf{if0}(\mathbf{1}^{-}(n), e_1, \\
&\quad \mathbf{if0}(\mathbf{2}^{-}(n), e_2 \\
&\quad \mathbf{3}^{-}(n))))
\end{aligned}
$$

$$
\begin{aligned}
pos(path, pr) &= \mathbf{if0}(\mathbf{1}^{-}(path), val(pr), \\
&\quad \mathbf{if0}(even?(path), \mathbf{cons}(pos(div2(path), pr), \perp), \\
&\quad \mathbf{cons}(\perp, pos(div2(path, prime)))))
\end{aligned}
$$

$$
\begin{aligned}
even?(path) &= \mathbf{if0}(path, path, \\
&\quad \mathbf{if0}(\mathbf{1}^{-}(path), path, even?(\mathbf{2}^{-}(path))))
\end{aligned}
$$

$$
\begin{aligned}
div2(path) &= \mathbf{if0}(path, path \\
&\quad \mathbf{if0}(\mathbf{1}^{-}(path), 0, \mathbf{1}^{+}(div2(\mathbf{2}^{-}(path)))))
\end{aligned}
$$

Given a computable element d, since there is a partial recursive function f corresponding to that element, we can build the constant function $\mathbf{d}$ such that $\mathbf{E}[\![\mathbf{d}]\!]$ is the element d.

Since any partial recursive function can be written in the language KL, the function f can be expressed in that language by the function $\mathbf{f}$. In that case, we can build the $\mathbf{d}$ in the following way.

$$
\begin{aligned}
g(n) &= \mathbf{if0}(\mathbf{f}(n), \mathbf{cons}(g(mult2(n)), g(\mathbf{1}^{+}(mult2(n))))), val(n)) \\
\\
mult2(n) &= \mathbf{if0}(n, n, \mathbf{1}^{+}(\mathbf{1}^{+}(mult2(\mathbf{1}^{-}(n))))) \\
\\
\mathbf{d} &= g(1)
\end{aligned}
$$

In addition to computable elements, the language must express all computable OS-functions if it is to be universal. An OS-function is computable if its graph, *i.e.*, the set $\{\vec{b}, f(\vec{b})\}$ where $\vec{b}$ are finite elements, is recursively enumerable. Since the

OS-functions can be embedded in $\mathbf{T}$ and interpreted by $Apply$, it is sufficient to show that KL can define all the computable elements of $\mathbf{T}$ to prove the universality of KL.

In conclusion, KL is a universal language for OS-domains. It is a first-order recursion equation style language with several possible evaluation strategies. It has a rewriting semantics that fully agrees with the denotational semantics.

6.3 A Universal Language for $\mathbf{T}^*$

The preceding discussion shows that KL serves as a universal language for $\mathbf{T}$. We can generalize KL to provide a language for the other universal domain $\mathbf{T}^*$. This section outlines one such generalization, **KL***, a computationally complete language for $\mathbf{T}^*$.

Recall that $\mathbf{T}^*$ is built by generalizing the construction of a node to provide for arbitrary arity. Therefore, **KL*** must provide constructs to build arbitrary arity trees. The obvious strategy is to provide one language construct for each of the domain constructs $\mathrm{list}_m()$. However, this strategy will not work for the following reasons.

Assume that there are language constructs $\mathbf{cons}_m$ that produce an m-ary tree for any m. Consider a function, say *build* that produces an m-ary tree with $\perp$'s at all the leaves for an input of m. Such a function is clearly computable, since under any computable enumeration, all the m-ary trees with $\perp$'s at nodes can be enumerated. However, we can prove that this function cannot be computed with tree building primitives $\mathbf{cons}_m$ as follows.

Any expression that generates an m-ary tree contains $\mathbf{cons}_m$; otherwise, it cannot generate such a tree. Similarly, if an expression e_f denotes a function that generates an expression containing $\mathbf{cons}_m$ on some input, then e_f must contain the expression $\mathbf{cons}_m$. Therefore any function that computes *build* must contain all the constructs $\mathbf{cons}_m$ for all m. Hence, it is impossible to write a finite expression denoting *build*.

Thus, we are forced to provide a generic construct **CONS**, that takes an argument n and provides an n-ary tree. Using such a tree building primitive, we can construct *build*. We can even formally prove that any computable element of the domain can be expressed with a language built around the construct **CONS**.

The language **KL*** is similar to KL. It is a first-order recursion schema with the following primitive constants. As usual, it includes a binary operator *Apply*: $\mathbf{T}^* \times \mathbf{T}^* \rightarrow_{os} \mathbf{T}^*$ and a set of constants: $\{0, \mathsf{error}_i, \mathbf{1}^+, \mathbf{1}^-, \mathsf{arity}, \mathsf{number?}, \mathsf{select}, \mathsf{insert}, \mathsf{construct}, \mathbf{if0}\}$. All the constants except 0 and error_i are elements of $\mathbf{T}^* \rightarrow_{os} \mathbf{T}^*$ (embedded in $\mathbf{T}$); they are interpreted as functions by *Apply*. For notational convenience we use them as bona fide uncurried functions, without showing the implicit *Apply*.

The primitive constants 0 and error_i provide the building blocks for constructing the trees over natural numbers and errors. The function $\mathbf{1}^+$ and $\mathbf{1}^-$ are the usual arithmetic operations that can be used to generate any natural number. The predicate $\mathsf{number?}$ distinguishes the numbers from trees. There is a usual conditional construct **if0** which behaves similar to the conditional in KL; it denotes the second argument if the first argument is 0, and third argument otherwise. All these functions except **if0** generate an error element if they encounter an error element as an argument. **if0** generates an error element if the first argument is an error element.

The functions $\mathsf{select}, \mathsf{insert}$, and $\mathsf{construct}$ deal with the variable arity trees. The function select selects a specified branch of a tree, if such a branch exists. Similarly, insert inserts a tree or a number at a specified branch of a tree. The function $\mathsf{construct}$ produces a tree with all $\perp$ leaves of specified arity.

The denotation semantics of these primitive operators can be given as follows:

As described in the beginning of this chapter, programs written in the first-order recursion equations over these functions are assigned denotations based on the preceding definitions.

The operational semantics for this language depends only on the δ-rules. We present the δ-rules as follows.

The constants:

$$
\begin{array}{rcl}
\mathbf{E}[\![0]\!]\sigma & = & 0 \\
\mathbf{E}[\![\mathsf{error}_i]\!]\sigma & = & \mathsf{error}_i
\end{array}
$$

Unary Operations

$$
\begin{array}{rcl}
\mathbf{E}[\![\mathbf{1^+}]\!]\sigma & = & \{n \rightarrow n+1\} \cup \{\mathsf{error}_i \rightarrow \mathsf{error}_i\} \\
\mathbf{E}[\![\mathbf{1^-}]\!]\sigma & = & \{0 \rightarrow 0\} \cup \{n+1 \rightarrow n\} \cup \{\mathsf{error}_i \rightarrow \mathsf{error}_i\} \\
\mathbf{E}[\![\mathsf{number?}]\!]\sigma & = & \{\mathrm{list}_m(\perp^m) \rightarrow 1\} \cup \{n \rightarrow 0\} \cup \{\mathsf{error}_i \rightarrow \mathsf{error}_i\} \\
\mathbf{E}[\![\mathsf{construct}]\!]\sigma & = & \{0 \rightarrow \mathrm{list}_\omega(\perp \ldots)\} \cup \{n \rightarrow \mathrm{list}_n(\perp^n)\} \cup \{\mathsf{error}_i \rightarrow \mathsf{error}_i\}
\end{array}
$$

Binary Operations

$$
\begin{array}{rcl}
\mathbf{E}[\![\mathsf{select}]\!]\sigma & = & \{n \times \mathrm{list}_m((\perp, \ldots, a_n, \ldots, \perp)) \rightarrow a_n\} \\
& & \cup\{\mathsf{error}_i \times x \rightarrow \mathsf{error}_i\} \\
& & \cup\{n \times \mathsf{error}_i \rightarrow \mathsf{error}_i\}
\end{array}
$$

Where $n \in \mathbf{N}^*$,$n \leq m$, and a_n is in n'th position at the node

Ternary Operations

$$
\begin{array}{rcl}
\mathbf{E}[\![\mathbf{if0}]\!]\sigma & = & \{0 \times x \times \perp \rightarrow x\} \cup \{1 \times \perp \times y \rightarrow y\} \cup \{\mathsf{error}_i \times \perp \times \perp \rightarrow \mathsf{error}_i\} \\
\mathbf{E}[\![\mathsf{insert}]\!]\sigma & = & \{(n \times x \times \mathrm{list}_m(\perp^m)) \rightarrow \mathrm{list}_m(\perp, \ldots, x, \ldots)\} \\
& & \cup\{\mathsf{error}_i \times x \times \mathrm{list}_m(\perp^m) \rightarrow \mathsf{error}_i\} \\
& & \cup\{n \times x \times \mathsf{error}_i \rightarrow \mathsf{error}_i\}
\end{array}
$$

Where $n \leq m$, and x is placed in the n'th position in the result

In the preceding definitions $n, m \in \mathbf{N}^*$, and $x, y \in \mathbf{T}^*$

Figure 6.3 Denotations of the Primitive Functions in **KL***

The proof of adequacy carries over for KL *mutatis mutandis*. Recall that to show adequacy, we must prove that denotational equivalence implies operational equivalence. In other words, the evaluation of the program diverges iff the denotation is $\perp$. As in the case of KL, we can define different evaluation strategies: Lazy evaluation strategy and Full evaluation strategy. Under either strategy the evaluation terminates iff the denotation is non-bottom.

The language **KL*** is computationally complete for $\mathbf{T}^*$. The proof can be adapted from the proof of universality of KL. The outline of the proof can be described as follows. It is easy to see that **KL*** can express all partial recursive functions over natural numbers. Also, we can define a standard one-to-one and onto mapping from

$$
\begin{aligned}
\mathsf{1}^+(n) &\longrightarrow n+1\\
\mathsf{1}^+(\mathsf{error}_i) &\longrightarrow \mathsf{error}_i\\
\mathsf{1}^-(0) &\longrightarrow 0\\
\mathsf{1}^-(n+1) &\longrightarrow n\\
\mathsf{1}^-(\mathsf{error}_i) &\longrightarrow \mathsf{error}_i\\
\mathsf{number?}(n) &\longrightarrow 0\\
\mathsf{number?}(\mathsf{List}_n()) &\longrightarrow 1\\
\mathsf{number?}(\mathsf{error}_i) &\longrightarrow \mathsf{error}_i\\
\mathsf{if0}(0)(a)(b) &\longrightarrow a\\
\mathsf{if0}(\mathsf{error}_i)(a)(b) &\longrightarrow \mathsf{error}_i\\
\mathsf{if0}(n+1)(a)(b) &\longrightarrow b\\
\mathsf{select}(n)(\mathsf{List}_m(a_1,\ldots,a_n,\ldots,a_m)) &\longrightarrow a_n\\
\mathsf{insert}(n)(a)(\mathsf{List}_m(a_1,\ldots,a_n,\ldots,a_m) &\longrightarrow \mathsf{List}_m(a_1,\ldots,a_n,\ldots a_m)[a_n/a]\\
\mathsf{construct}(n) &\longrightarrow \mathsf{List}_n(\omega,\omega,\ldots(\text{n times}))
\end{aligned}
$$

Figure 6.4 δ-rules for **KL***

$\mathbf{T}^*$ into a subset of the binary trees of $\mathbf{T}^*$. Let that function be f. It is easy to show that f and its inverse are expressible in **KL***.

Let x be a computable tree in $\mathbf{T}^*$. Then $f(x)$ is a computable binary tree, which can be expressed as a partial recursive function g_x. By using similar functions as in the case of KL, we can build the binary tree $f(x)$ from g_x. Finally, we can build x by applying f^{-1} to that. Therefore, from a partial recursive function describing the binary form of an n-ary tree, we can build the m-ary tree. The functions construct, insert, and select are essential for defining f^{-1}.

Chapter 7

Categorical Model for SPCF

In the previous chapters, we studied universal domains and the corresponding languages. In this chapter, we focus on the problem of defining a fully abstract, computationally complete semantics for SPCF, a sequential functional language extending Plotkin's language PCF. We will present both an operational and a denotational semantics for SPCF; the latter heavily relies on OS-domains and OS-functions. Our most important result is that SPCF is *computationally complete* for our denotational model. We will not present the motivation behind the design of SPCF, which has been presented in [CF92].

Plotkin introduced PCF in [Plo77], a call-by-name idealized functional programming language based on lambda calculus with constants such as integers, booleans, and functions to compute over those constants. In the literature, PCF has been extensively studied with respect to the full-abstraction problem. The traditional model based on continuous functions is not fully abstract for PCF because it contains deterministic parallel operations like *parallel-or*.

Cartwright and Felleisen constructed the first fully abstract model for SPCF [CF92]. Unlike the previous attempts at full abstraction, which either add parallel constructs to PCF or rely on non-effective, non-algebraic quotient constructions, their solution relies of adding *sequential* constructs to PCF and modifying the denotations to reflect the sequentiality of the language.

The design of SPCF and its fully abstract model was based on two observations: (i) programmers can observe the order of evaluation in procedures through the propagation of errors, and (ii) with suitable control operators, *programs* can determine the order of evaluation of procedure arguments. The full details of the language are available in [CF92, CCF94, KCF93]. In [Cur92], Curien shows that Cartwright and Felleisen's model is isomorphic to a cartesian closed category of sequential algorithms enriched with error-elements. In this chapter, we construct a more "abstract" model for SPCF based on OS-functions instead of concrete decision tree (sequential algorithm) representations for functions. Our model is isomorphic to both Cartwright and Felleisen's model and Curien's model, but the abstract presentation makes it much easier to prove computational completeness, yielding a proof of full abstraction as a trivial corollary.

The rest of the chapter is organized as follows. First, we present the language SPCF, and its rewriting rules. We briefly discuss the language and present the operational semantics based on the rewriting rules presented. Later, we show that the adequacy of the model, *i.e.*, the denotational equivalence implies the operational equivalence. Finally, we show that SPCF is computationally complete for its domains, from which we will derive the full abstraction result.

7.1 SPCF: Syntax and Semantics

SPCF is a *typed* lambda-calculus based language. We present the well-formed phrases of SPCF in 7.1.

The first part of Figure 7.1 defines the set of syntactically well-formed SPCF phrases. An SPCF phrase M is either a constant (i.e., a numeral $\ulcorner n \urcorner$, $n \geq 0$, one of two error constants, error_1 and error_2, or a functional constant), a typed variable x^τ, a λ-abstraction, or an application.

The second part of the Figure 7.1 presents the possible types. They consist of a ground type o and infinite number of finitely generated types of the form $\sigma \to \tau$.

The typing rules are presented in the last part of the Figure 7.1. They are similar to the typing rules of PCF except for the last line which defines the typing rules for the additional constructs of SPCF.

Syntax:

$$
\begin{array}{rcl}
M & ::= & c \mid x \mid (\lambda x.M) \mid (M\ M) \\
c & ::= & \ulcorner n \urcorner \mid \mathsf{error}_1 \mid \mathsf{error}_2 \mid \mathsf{1}^+ \mid \mathsf{sub1} \mid \mathsf{if0} \mid \mathsf{Y}^\tau \mid \mathsf{is\text{-}const?} \\
x & ::= & x^\tau \mid y^\tau \mid \ldots
\end{array}
$$

Types: $\sigma, \tau ::= o \mid (\tau \to \sigma)$

Type Checking:

$$\frac{A, x^\tau \vdash M : \tau'}{A \vdash \lambda x^\tau.M : \tau \to \tau'} \qquad \frac{A \vdash M : \tau' \to \tau; \quad A \vdash M' : \tau'}{A \vdash (MM') : \tau}$$

$$
\begin{array}{lll}
A \vdash x^\tau : \tau \text{ if } x^\tau \in A & A \vdash \ulcorner n \urcorner : o \text{ for all } n & A \vdash \mathsf{Y}^\tau : (\tau \to \tau) \to \tau \\
A \vdash \mathsf{1}^+ : o \to o & A \vdash \mathsf{sub1} : o \to o & A \vdash \mathsf{if0} : o \to o \to o \to o \\
A \vdash \mathsf{is\text{-}const?} : (o \to o) \to o & A \vdash \mathsf{error}_1 : o & A \vdash \mathsf{error}_2 : o
\end{array}
$$

Figure 7.1 SPCF: Syntax

7.1.1 Operational Semantics

Figure 7.2 presents the operational semantics. This presentation follows the style used in [Fel87]. First, we present a scheme of rewriting rules. These rewriting rules transform one phrase into another. Later, we use these rewriting rules to provide an evaluation of the programs.

According to the rewriting rules in Figure 7.2, the meaning of phrases in SPCF is the usual one, *i.e.*, numerals and functional constants have their expected behavior, λ-abstractions are call-by-name procedures, and juxtaposition is function application.

The error constants, error_1 and error_2, generate special "error" values that are propagated according to the usual by-need evaluation order in call-by-name programs. If an SPCF procedure uses an argument and the by-need evaluation of the argument generates an error value e, the procedure returns the value e. On the other hand, if

Evaluation Contexts:

$$\begin{array}{lll} E & ::= & [\ ] \mid (c^1\ E) \mid (\mathsf{if0}\ E\ M\ M) \mid (\mathsf{is\text{-}const?}\ (\lambda x.E)) \mid (E\ M) \\ c^1 & ::= & \mathbf{1}^+ \mid \mathsf{sub1} \end{array}$$

Rewriting Rules:

$$\begin{array}{rcll} (\lambda x.M)\ M' & \longrightarrow & M[x/M'] & (\beta) \\ \mathsf{Y}\ M & \longrightarrow & M\ (\mathsf{Y}\ M) & (\mathsf{Y}) \\ \\ (\mathbf{1}^+\ \ulcorner n \urcorner) & \longrightarrow & \ulcorner n+1 \urcorner & (\mathbf{1}^+) \\ (\mathsf{sub1}\ \ulcorner n+1 \urcorner) & \longrightarrow & \ulcorner n \urcorner & (\mathsf{sub1}) \\ (\mathsf{if0}\ \ulcorner 0 \urcorner\ M\ N) & \longrightarrow & M & (\mathsf{if0}_0) \\ (\mathsf{if0}\ \ulcorner n+1 \urcorner\ M\ N) & \longrightarrow & N & (\mathsf{if0}_1) \\ E[\mathsf{error}_1] & \longrightarrow & \mathsf{error}_1 & (\mathsf{error}_1)\ E \text{ not empty} \\ E[\mathsf{error}_2] & \longrightarrow & \mathsf{error}_2 & (\mathsf{error}_2)\ E \text{ not empty} \\ \mathsf{is\text{-}const?}(\lambda x.\ulcorner n \urcorner) & \longrightarrow & \ulcorner 0 \urcorner & (\mathsf{const}) \\ \mathsf{is\text{-}const?}(\lambda x.E[x]) & \longrightarrow & \ulcorner 1 \urcorner & (\mathsf{strict}) \\ & & & \text{if } x \text{ is not captured by } E \end{array}$$

Figure 7.2 SPCF: Rewriting Rules

the procedure ignores an argument, the meaning of the argument is irrelevant because by-need evaluation never evaluates the argument.

One of the interesting functions of SPCF is is-const?. It is a predicate on functions that checks if the given function is a constant function. Though is-const? looks simple, it is as expressive as other, more realistic control structures such as catch and call/cdc, which we will describe later.

To illustrate the use of error elements and is-const? for observing the sequentiality of procedures, let us consider the following two definitions of a binary addition procedure in SPCF:

$$+_l \;=\; \mathsf{Y}(\lambda f\,.\,(\lambda xy\,.\,\mathsf{if0}\ x\ y\ (\mathbf{1}^+\ (f\ (\mathsf{sub1}\ x)\ y))))$$

$$+_r \quad = \quad \mathsf{Y}(\lambda f\,.\,(\lambda xy\,.\,\mathsf{if0}\ y\ x\ (\mathbf{1}^+\ (f\ x\ (\mathsf{sub1}\ y)))))$$

The first version $+_l$ recurs on the first argument (x); the second version recurs on the second argument (y). In languages without errors and control operators, both procedures are observationally equivalent. In SPCF, we can distinguish between the two versions by applying to error-values: $(+_l\ \mathsf{error}_1\ \mathsf{error}_2)$ produces error_1 whereas $(+_r\ \mathsf{error}_1\ \mathsf{error}_2)$ produces error_2. Thus, the *user* can observe the difference in these procedures. Using is-const?, a *program* can exploit the difference in order of evaluation as follows:

$$(\lambda\ f.\ (\mathsf{is\text{-}const?}\ (\lambda\ x.\ (\mathsf{if0}\ (\mathsf{is\text{-}const?}\ (\lambda\ y.\ (f\ x\ y)))\ \ulcorner 1\urcorner\ \ulcorner 1\urcorner))))$$

The above procedure yields $\ulcorner 0\urcorner$ if we apply it to $+_l$; it yields $\ulcorner 1\urcorner$ if we apply it to $+_r$.

Functional and Syntactic Extensions

When we write SPCF programs, we freely use functions and syntactic abbreviations that are easily definable in the language. Specifically, we will assume that the names $+$ and $-$ denote some addition and subtraction functions. When we use these names, the evaluation order of the arguments for these functions is unimportant. We will also use the following syntactic abbreviations:

$$\begin{array}{rcl}
\Omega & \stackrel{df}{=} & (\mathsf{sub1}\ x) \\
\mathbf{let}\ x = M\ \mathbf{in}\ N & \stackrel{df}{=} & ((\lambda x\,.\,N)\ M) \\
\mathbf{let}{*}\ x = M, y = K\ \ldots\ \mathbf{in}\ N & \stackrel{df}{=} & ((\lambda x\,.\,\mathbf{let}{*}\ y + K \ldots\ \mathbf{in}\ N)\ M) \\
\mathbf{let}{*}\ x = M\ \mathbf{in}\ N & \stackrel{df}{=} & ((\lambda x\,.\,N)\ M) \\
\mathbf{letrec}\ L = M\ \mathbf{in}\ (L\ N) & \stackrel{df}{=} & \mathsf{Y}\ (\lambda L\,.\,M)\ N
\end{array}$$

and some occasional syntactic extensions that are self-explanatory.

Higher-order Constants

The version of SPCF we outlined above differs from SPCF as given in [CF92, KCF93] in some details. The original SPCF has a family of constructs called catch^τ of type $\tau \to o$ for every type τ. However, the semantics of catch depends only on the number of inputs in its type, *i.e.*, k where

$$\tau \to o = \tau_1 \to (\tau_2 \to (\ldots(\tau_k \to o))).$$

We refer to a catch procedure with k inputs as catch_k, and give its semantics as follows:

$$\begin{aligned} \mathsf{catch}^o(x) &\longrightarrow x \\ \mathsf{catch}_k(\lambda x_1, x_2 \ldots x_k.E[x_i]) &\longrightarrow \ulcorner i-1 \urcorner \\ \mathsf{catch}_k(\lambda x_1, x_2 \ldots x_k.\ulcorner n \urcorner &\longrightarrow \ulcorner k+n \urcorner \end{aligned}$$

where the evaluation contexts are extended to include

$$E ::= \ldots \mid (\mathsf{catch}(\lambda x_1 \ldots x_k.E)).$$

Subsequent papers [KCF93] introduced $\mathsf{call/cdc}$, a construct similar to Scheme's $\mathsf{call/cc}$ procedure. $\mathsf{call/cdc}$ has the type $((o \to o) \to o) \to o$. It applies the current continuation to its argument, but it can be used only once. Since SPCF does not have state, we cannot use the continuations outside the lexical scope. The operational semantics of $\mathsf{call/cdc}$ is:

$$\begin{aligned} \mathsf{call/cdc}(\lambda f.\ulcorner n \urcorner) &\longrightarrow \ulcorner n \urcorner \\ \mathsf{call/cdc}(\lambda f.E[f\ulcorner n \urcorner]) &\longrightarrow \ulcorner n \urcorner \end{aligned}$$

where the evaluation contexts are extended to include

$$E ::= \ldots \mid (\mathsf{call/cdc}(\lambda x.E)).$$

Even though catch is presented as an exception catching construct, and $\mathsf{call/cdc}$ is presented as a control construct, they are equivalent to the higher-order predicate is-const?. In other words, using local syntactic transformations, we can express any two of these constants using the third constant. Therefore, these three constructs are equally expressive [Fel90].

Proposition 7.1 The three constructs is-const?, catch, and call/cdc are interdefinable.

Proof.

is-const? **in terms of** catch is-const? can trivially be defined as:

$$(\lambda\ f.\ (\mathsf{if0}\ (\mathsf{catch}^{o\to o}\ f)\ 0\ 1))$$

catch **in terms of** call/cdc: Given call/cdc we can write catch as follows. This function passes the arguments such that when they are evaluation, they invoke the continuation. For type correctness, we use $\mathsf{if0}^{(o\to\tau_1\to\tau_1\to\tau_1)}$. It is easy to write such an if0 using $\mathsf{if0}^{(o\to o\to o\to o)}$ and lambda abstraction.

$$\begin{aligned}
\mathsf{catch}_0 &= \lambda x\,.\,x\\
\mathsf{catch}_k &= \lambda f\,.\,\mathsf{call/cdc}(\lambda t.(+\ \ulcorner k\urcorner\ (f\\
&\qquad\qquad (\mathsf{if0}\ (t\ \ulcorner 0\urcorner)\ \Omega^{\tau_1}\ \Omega^{\tau_1})\\
&\qquad\quad \ldots(\mathsf{if0}\ (t\ \ulcorner k-1\urcorner)\ \Omega^{\tau_k}\ \Omega^{\tau_k}))))
\end{aligned}$$

where the subscript k indicates the number of inputs to catch_k and the type of catch_k is $(\tau_1\to(\tau_2\to(\ldots(\tau_k\to o))))$.

It is easy to verify that the above definitions pass type checking. If the rewriting ever evaluates the argument x_i, then the function $(\mathsf{if0}\ (t\ \ulcorner i\urcorner)\ \Omega^{\tau_i}\ \Omega^{\tau_i})$ is reduced after application. Eventually, call/cdc returns the argument $\ulcorner i-1\urcorner$. If the function does not evaluate any of its arguments, then it returns the result after adding $\ulcorner k\urcorner$.

call/cdc **in terms of** is-const?: Writing call/cdc using is-const? is more involved than the previous item. The following function F defines call/cdc in terms of is-const?.

Given a function $g = \lambda\ h.e$ of type $(o \rightarrow o) \rightarrow o$, it checks to see if g invokes h, its argument function. If it does not, then g is invoked with $\Omega^{o \rightarrow o}$. Therefore, $F(\lambda\ h.\ulcorner n \urcorner) = n$, confirming to the behavior of call/cdc. If g is not a constant function, then g invokes its argument function with some natural number. To check which number is passed to h, we check for each number in a loop. If h is invoked with some number, then the loop terminates with that answer.

```
(λ f. (if0 (is-const? (λ x. (f (λ y. x))))
      (f Ω^{o→o})
      (letrec
       L = (λ v. (if0 (is-const?
                       (λ x. (if0 (is-const?
                                    (λ y. (f (λ z. (=? v z) x y))))
                                   1
                                   1)))
                       (L (1+ v))
                       v))
         in
       (L ⌜0⌝))))
```

Thus, all the three constructs are equally expressive. □

Evaluation Of Programs

The evaluation of a program makes use of the rewriting rules and the evaluation contexts presented in the figure 7.2. According to these rules, a program can be decomposed into an evaluation context and a redex. A redex is a subterm that matches the left hand side of some primitive rewriting rules. The appropriate rewriting rule transforms the redex giving a new program. Thus, a single evaluation step is defined as:

$$E[M] \;\triangleright\; E[M'] \text{ iff } M \longrightarrow M'$$

We evaluate a program by continuing the stepping process until the program reduces to a ground constant. This process of evaluation is defined by the partial function *eval* from programs to ground constants:

$$eval(M) = n \iff M \;\triangleright^*\, n$$

Programs that fail to terminate are said to be *divergent*. A simple example of a program that does not terminate is (`sub1` 0).

If *eval* has to be a function, it must evaluate a program to at most one constant. We will prove it by showing that at any stage in the evaluation there is a unique way the evaluation can proceed. In fact, the evaluation proceeds by reducing the left-most, outer-most application.

Lemma 7.1 Every SPCF application can be uniquely decomposed into an evaluation context and a redex.

Proof. Proof proceeds by structural induction on the terms.

Case $(M\ N)$**:** If M is lambda abstraction, then the whole expression is a redex with the empty evaluation context. If M is itself as application then, by IH, $M = E[M']$, there by producing an evaluation context $(E[\ ]\ N)$ and a redex M'. This context is unique because of IH.

Case $(c^1 M)$**:** Since $c^1 \in \{\mathbf{1^+}, \mathtt{sub1}\}$, if M is a numeral, or an error value, then the evaluation context is $(c^1\ [\ ])$ by the definition of evaluation contexts. If M is not a numeral then by IH M can be uniquely divided into an evaluation context and a redex, as M must be an application.

Case ($\mathsf{if0}$ $M\ N\ P$)**:** It is similar to the preceding case of $\mathbf{1^+}$.

Case (Y M)**:** Here M is the redex and the rest is the evaluation context.

Case (is-const? $\lambda x.M$)**:** If M is a numeral, or an error value, then the whole expression is a redex with empty evaluation context. If M is an application, then by IH, we can divide into evaluation context and redex $E[M']$, there by providing an evaluation context (is-const? $\lambda x.E[\,]$) and a redex M'.

□

Using the preceding lemma, we can prove that *eval* is a partial function. That means, a program whenever it terminates, evaluates to only one value. However, some programs may fail to terminate.

Lemma 7.2 *eval* is a partial function on the programs of SPCF into $\{0, 1, \ldots,$ error_1, $\mathsf{error}_2\}$.

Proof. Since program is of type o, it is either a numeral or an error value or an application. If is a numeral or an error value, the partial function *eval* terminates with that value as the answer. If it is an application, we can find an evaluation context and a redex uniquely. Since at any stage the reduction retains the type of the program, we get the next step uniquely. Also, a redex can be reduced using only of the rewriting rules. Therefore, the evaluation, if it terminates, always produces the same answer. □

7.1.2 Denotational Semantics

Using the framework of $\mathcal{OS}$, the ccc of OS-domains, we can provide a fully abstract environment model [Mey82] for SPCF. This model is formally defined in Figure 7.3. It maps each type τ into an OS-domain $\mathcal{T}[\![\tau]\!]$. Specifically, it maps the ground type o to $\mathbf{N}_\perp^E$, and uses OS-function for function domains. As shown in Chapter 4, it is the standard method to give semantics for a lambda-calculus based language.

An *environment* ρ is an observably sequential function mapping the flat OS-domain $\mathcal{V}_\perp^E = \mathcal{V} \cup \mathbf{E}_\perp$ of SPCF variable names to the union *Dom* of all domains

$\mathcal{T}[\![\tau]\!]$ that respects the types of variable names: for all $x^\tau \in \mathcal{V}$, $\rho(x^\tau) \in \mathcal{T}[\![\tau]\!]$. It is easy to prove that the set of environments forms an OS-domain $\mathcal{E}$ under the pointwise approximation ordering. The function $\mathcal{C}$ maps constants of type τ to denotations in $\mathcal{T}[\![\tau]\!]$. Finally, if M is of type τ, the meaning function $\mathcal{M}$ assigns M the denotation $\mathcal{M}[\![M]\!] \in \mathcal{E} \to \mathcal{T}[\![\tau]\!]$.

The notation used in the definition follows the standard conventions in the literature. In particular, for a closed term M, we simply write $\mathcal{M}[\![M]\!]$ to denote the meaning of the term M—instead of $\mathcal{M}[\![M]\!]\rho$—because the environment ρ is irrelevant. By the same token, we write $\mathcal{M}[\![M]\!]$ to denote the meaning of an SPCF program M, since it is closed term of type o.

The only requirement for this model to be a valid categorical model is that the constant functions must belong to a domain in the category. It can be easily seen that all the first-order functions are OS-functions. The higher-order predicate is-const? is an OS-function: it maps a constant function to 0, and a non-constant function to 1. If the function passed is an error generating function, it maps it to the appropriate error element.

Given the two kinds of semantics, operational and denotational, we must show that they correspond naturally. We show that if two phrases are denotationally equivalent, then they must be operationally equivalent. We define these concepts as follows.

Definition 7.2. (*Observational Equivalence*) Let M and N be two SPCF terms of type τ. Then, M and N are observationally equivalent if for all contexts C^τ (programs with holes in places of terms of type τ) such that $C[M]$ and $C[N]$ are programs,

$$\mathcal{M}[\![C[M]]\!] = \mathcal{M}[\![C[N]]\!].$$

We write $M \simeq_{SPCF} N$ if M and N are observationally equivalent. ∎

The preceding definition states that if two phrases are observationally equivalent, they generate the same answer in all program contexts. That is, an observer cannot differentiate these phrases based on their behavior.

Domains

$$
\begin{array}{rcl}
\mathcal{T}[\![o]\!] & = & \mathbf{N}_{\perp}^{E} \\
\mathcal{T}[\![\sigma \to \tau]\!] & = & [\mathcal{T}[\![\sigma]\!] \to_{os} \mathcal{T}[\![\tau]\!]] \\
\mathcal{V}_{\perp}^{E} & = & \mathcal{V} \cup \mathbf{E}_{\perp} \\
\mathcal{E} & = & \{f \in \mathcal{V}_{\perp}^{E} \to_{os} \bigcup_{\tau} \mathcal{T}[\![\tau]\!] \mid f(x^{\tau}) \in \mathcal{T}[\![\tau]\!]\}
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathbf{N}_{\perp}^{E} & \stackrel{df}{=} & \mathbf{N} \cup \mathbf{E}_{\perp} \\
\mathbf{E}_{\perp} & \stackrel{df}{=} & \{\perp, \mathsf{error}_1, \mathsf{error}_2\}
\end{array}
$$

Semantic functions

$$
\begin{array}{rcl}
\mathcal{C} & : & \mathit{Constants}_{\tau} \to \mathcal{T}[\![\tau]\!] \\
\mathcal{C}[\![\ulcorner n \urcorner]\!] & = & n \\
\mathcal{C}[\![\mathsf{error}_i]\!] & = & \mathsf{error}_i \\
\mathcal{C}[\![\mathsf{1}^+]\!] & = & \{(\perp, \perp), (\mathsf{error}_i, \mathsf{error}_i), (l, l+1) \mid l \in \mathbf{N}\} \\
\mathcal{C}[\![\mathsf{sub1}]\!] & = & \{(\perp, \perp), (\mathsf{error}_i, \mathsf{error}_i), (0, \perp), (l+1, l) \mid l \in \mathbf{N}\} \\
\mathcal{C}[\![\mathsf{if0}]\!] & = & \{(\perp, l, m, \perp), (\mathsf{error}_i, l, m, \mathsf{error}_i), (0, l, m, l), (k+1, l, m, m) \mid l, m, k \in \mathbf{N}\} \\
\mathcal{C}[\![\mathsf{Y}]\!] & = & \underline{\lambda} f^{(\sigma \to \tau) \to (\sigma \to \tau)}. \bigsqcup_{n=0}^{\infty} f^{n}(\perp^{(\sigma \to \tau)}) \\
\mathcal{C}[\![\mathsf{is\text{-}const?}]\!](f) & = & \begin{cases} 0 & (\perp, n) \in f \\ 1 & (\mathsf{error}_i, \mathsf{error}_i) \in f \\ \mathsf{error}_i & (\perp, \mathsf{error}_i) \in f \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{M} & : & \mathit{Terms}_{\tau} \to_{os} \mathcal{E} \to_{os} \mathcal{T}[\![\tau]\!] \\
\mathcal{M}[\![b]\!]\rho & = & \mathcal{C}[\![b]\!] \\
\mathcal{M}[\![x]\!]\rho & = & \rho[\![x]\!] \\
\mathcal{M}[\![\lambda x.M]\!]\rho & = & (\underline{\lambda} v.\mathcal{M}[\![M]\!]\rho[x/v]) \\
\mathcal{M}[\![(M\ N)]\!]\rho & = & \mathcal{M}[\![M]\!]\rho(\mathcal{M}[\![N]\!]\rho)
\end{array}
$$

Figure 7.3 The semantics of SPCF

The property of the *OS*-domains model for SPCF is that two terms map to the same denotation iff they are observationally equivalent. It is obvious that for a denotational semantics to be useful, denotational equivalence must imply observational equivalence. This property is called *adequacy*. The property that the observational semantics imply denotational semantics is called *full abstraction*.

Adequacy

To show that the *OS*-domains model is adequate for the operational semantics of SPCF, we need to show some lemmas. We can naturally divide the proof of adequacy into two parts.

1. If $M \rhd N$, then $\mathcal{M}[\![M]\!] = \mathcal{M}[\![N]\!]$.

2. If the evaluation of the program M diverges, then $\mathcal{M}[\![M]\!] = \bot$.

Together these two statements imply adequacy of the model.

As with KL in Chap. 6, we introduce Ω into the language to make *eval* a total function. Ω is meant to capture a "stuck" state or divergence. In addition we introduce a series of Y_i whose limit point is Y. Each Y_i expand the recursion depth only up to i levels, and replaces the last invocation of the function with Ω.

Definition 7.3. (Ω *rules*) We extend SPCF to include Ω and Y_i as a part of the language. The operational rules are extended to include:

$$
\begin{array}{rcll}
E[\Omega] & \longrightarrow & \Omega_o & \qquad (\Omega_0) \\
(\mathsf{sub1}\ 0) & \longrightarrow & \Omega_o & \qquad (\Omega_1) \\
(\mathsf{Y}_0\ M) & \longrightarrow & \Omega & \qquad (\mathsf{Y}_0) \\
(\mathsf{Y}_i\ M) & \longrightarrow & (M\ (\mathsf{Y}_{i-1}\ M)) & \qquad (\mathsf{Y}_1)
\end{array}
$$

We use Ω of appropriate type, to ensure the type correctness of the program. ∎

It is obvious that Y_i expands the function up to depth i. Also, Ω represents the divergent computation or a stuck state. We can define $eval_\Omega$ to include the new Ω rules while rewriting the program. Under this function, as usual, $E[M] \rhd E[N]$ if $M \longrightarrow N$, where $\longrightarrow$ is extended to Ω rules.

It is also easy to show that denotationally the following equations hold:

$$
\begin{array}{rcl}
\mathcal{C}[\![\Omega_o]\!] & = & \bot \\
\mathcal{C}[\![\Omega_\tau]\!] & = & \bot_\tau \\
\mathcal{C}[\![\mathsf{Y}_i]\!] & = & \underline{\lambda} f^{(\sigma\to\tau)\to(\sigma\to\tau)}. \bigsqcup_{n=0}^{i} f^n(\bot^{(\sigma\to\tau)})
\end{array}
$$

Now, we introduce a series of programs e_i related to e. These programs are related to e such a way that, $eval(e) = c$ iff there is an i for which $eval_\Omega(e_i) = c$. We will also show that the evaluation of e_i's always terminate. If $eval(e)$ diverges, then $eval_\Omega(e_i)$ is Ω for all i. We relate $eval_\Omega(e_i)$'s to the denotations and show the adequacy result.

These programs are defined by restricting the length of recursion using Y. By replacing Y with Y_i, we can obtain these programs:

$$e_i = e[\mathsf{Y}/\mathsf{Y}_i]$$

First, we prove the required relationship between e_i and e. If $eval(e)$ terminates with an answer c, then there must be an i for which $eval_\Omega(e_i) = c$. It means, that the evaluation of e unwinds the application of Y at most i times.

Lemma 7.3 For a program e, $eval(e) = c$ iff there exists i such that $eval_\Omega(e_i) = c$.

Proof. If $eval(e) = c$, then the redex in the evaluation context can never be ($\mathsf{sub1}$ 0). In addition, there is an upper bound on the number of time the rule (Y) is used. Let i be such an upper bound. We can show that $eval_\Omega(e_i) = c$ as follows. Following the rewriting of e under $eval$, we can rewrite e_i under $eval_\Omega$. The evaluation context, at no time, contains Ω, since $eval$ never encounters ($\mathsf{sub1}$ 0) or expands beyond the depth i. Therefore, since the reduction rules of $eval_\Omega$ uses are the same as $eval$ at every stage; hence $eval_\Omega(e_i)$ terminates with c.

If $eval_\Omega(e_i) = c$ for some i and $c \neq \Omega$, then the evaluation of e must yield c. It is easy to see that the evaluation of e_i never invokes any Ω rules; otherwise $eval_\Omega(e_i) = \Omega$. Since reduction of the Y_i rules is equivalent to the reduction of Y as long as Ω is not evaluated, the evaluation of e results in c. $\square$

Notice that c cannot be equal to Ω as $eval$ operates only on the SPCF phrases that do not contain Ω. According to the preceding lemma, if there is no i for which $eval_\Omega(e_i) = c$, then $eval$ cannot terminate with an answer.

Now, we must show that there cannot be two different programs e_k, e_l, both derived from e by suitable substitutions, evaluating to different non-Ω constants. However, the following lemma relates e_i's to their denotations and eventually to the denotation of e.

Lemma 7.4 If $eval_\Omega(e_i) = c$ then $\mathcal{M}[\![e_i]\!] = \mathcal{M}[\![c]\!]$. Here c can be a ground constant including Ω.

Proof. To show this lemma, it is sufficient to prove that if $M \rhd N$, then $\mathcal{M}[\![M]\!] = \mathcal{M}[\![N]\!]$. Since $e_i \rhd^* c$, we an infer $\mathcal{M}[\![e_i]\!] = \mathcal{M}[\![c]\!]$.

Let $M \rhd N$. Therefore, $E[M'] \rhd E[N']$, *i.e.*, M' reduces to N' by one of the rewriting rules. Since $E[M'] = ((\lambda x.E[x])M')$, it can easily be seen that $\mathcal{M}[\![M]\!] = \mathcal{M}[\![N]\!]$ iff $\mathcal{M}[\![M']\!]\rho = \mathcal{M}[\![N']\!]\rho$, for all ρ. Therefore the proof obligation can be reduced to showing

$$M \longrightarrow N \Rightarrow \forall\rho(\mathcal{M}[\![M]\!]\rho = \mathcal{M}[\![N]\!]\rho)$$

Proof proceeds by case analysis of the different rewriting rules.

(β): Let $M = ((\lambda x.M_1)M_2)$. Then $M \longrightarrow M_1[x/M_2]$. By definition, $\mathcal{M}[\![M]\!]\rho$ is equal to $\mathcal{M}[\![M_1]\!]\rho[x/\mathcal{M}[\![M_2]\!]\rho]$. It is easy to show by structural induction that it is equal to $\mathcal{M}[\![M_1[x/M_2]]\!]\rho$. The only tricky point is to see that the free variables of M_2 do not get closed by substituting in M_1.

($\mathbf{1}^+$), ($\mathsf{sub1}$), ($\mathsf{if0}_0$), ($\mathsf{if0}_1$): These rewriting rules are satisfied the denotational model, since the corresponding functions are given appropriate denotations.

(error_1), (error_2): Consider the evaluation contexts. If the evaluation is $(c^1 E)$, then the reduction satisfies the denotation of the basic functions. It works similarly in the case of $\mathsf{if0}$. Consider the case of $\mathsf{is\text{-}const?}$. If the redex is an error, the whole evaluation context generates an error, since $((\bot, \mathsf{error}_i), \mathsf{error}_i) \in \mathcal{C}[\![\mathsf{is\text{-}const?}]\!]$.

(const), (strict): Since ($\mathsf{is\text{-}const?}$ $\lambda x.\ulcorner n \urcorner$) $= 0$ by the denotational definition, (const) works. Similarly, if the phrase is ($\mathsf{is\text{-}const?}$ $(\lambda x.E[x])$) then $(\mathsf{error}_i, \mathsf{error}_i) \in \mathcal{M}[\![(\lambda x.E[x])]\!]\rho$, therefore, yields 1 as per the denotation.

Ω **rules:** The case of $\mathcal{M}[\![E[\Omega]]\!]\rho = \perp$ can be shown by case analysis of evaluation contexts. All these contexts are strict; they produce $\perp$ when they are presented with $\perp$ fill the hole in the context. It is easy to see that (sub1 0) denotes $\perp$ verifying the rule (Ω_1). Similarly, the rules $(\mathsf{Y}_0), (\mathsf{Y}_1)$ verify the denotations of $\mathsf{Y}_i, i \in \mathbf{N}$.

□

To show that different e_i's cannot evaluate to different non-Ω constants, we appeal to their denotations. Since the denotations of the answers are drawn from a flat domain, we can prove that the sequence can converge to only one ground constant, if we prove that the denotations form a chain.

Lemma 7.5 For a program e and the derived programs e_i, the following equality holds:

$$\mathcal{M}[\![e]\!] = \bigsqcup \mathcal{M}[\![e_i]\!]$$

Proof. The the proof proceeds by showing that $\{\mathcal{M}[\![e_i]\!] \mid i \in \mathbf{N}\}$ form a chain. Later, we show the equality.

To show that the denotations form a chain, we appeal to more general result: If $\mathcal{M}[\![M]\!]\rho \sqsubseteq \mathcal{M}[\![N]\!]\rho$ then $\mathcal{M}[\![C[M]]\!]\rho \sqsubseteq \mathcal{M}[\![C[N]]\!]$. This result to easy to prove by breaking the program $C[M]$ into $((\lambda x.C[x])\ M)$ where x is not bound in $C[\,]$. By the continuity of the model, we have the required result.

Since $\mathsf{Y}_i \sqsubseteq \mathsf{Y}_{i+1}$, we can break the program into a context (not evaluation context) containing Y. Therefore, $e_i = C[\mathsf{Y}_i]$ and $e_{i+1} = C[\mathsf{Y}_{i+1}]$. By the preceding result, we can say that $\mathcal{M}[\![e_i]\!] \sqsubseteq \mathcal{M}[\![e_{i+1}]\!]$. By a similar argument, we can show that $\mathcal{M}[\![e]\!] = \bigsqcup \mathcal{M}[\![e_i]\!]$. □

According to Lemma 7.4, $eval_\Omega$ of a term e_i is equal to $\mathcal{M}[\![e_i]\!]$ when it terminates. This lemma does not say what happens when $eval_\Omega$ diverges. The following lemmas show that $eval_\Omega$ can never diverge; hence the evaluation of e_i is always equal to its denotation, since it always converges to a constant.

We show that e_i terminates for all i by induction on i. Since the base case for 0 is quite elaborate, we present it as a separate lemma as follows.

Lemma 7.6 For any program e, the evaluation of e_0 always terminates.

Proof. For this proof, we employ a technique from [Plo77], where it is used to show the adequacy of PCF. Define a set $Norm_\tau$ for each τ, a set of phrases of type τ, as follows. Wherever we omit the subscript type τ for $Norm$, the appropriate type should be taken.

1. A term of type o, *i.e.*, a program, is in $\mathbf{Norm}_o$ iff it terminates under $eval_\Omega$.

2. An open phrase M is in $\mathbf{Norm}_\tau$ iff all the closed instantiations with terms from **Norm** belongs to $\mathbf{Norm}_\tau$.

3. A function M is in Norm iff the following statement holds:

$$N \in \mathbf{Norm} \Rightarrow (M\ N) \in \mathbf{Norm}$$

A term in **Norm** is said to be *normalizable*.

Now we prove that all the terms belong to $Norm$. Then, by the definition of **Norm**, all the programs terminate under $eval_\Omega$. The proof proceeds by induction on the structure of terms.

Variable x: Its instantiation with a term from **Norm** is in **Norm**. Hence it is in **Norm**.

Constants: All the basic constants such as the integers, Ω, and $\mathsf{error}_1, \mathsf{error}_2$ are normalizable.

It is easy to see all the function constants are normalizable. We will show for two cases.

Consider the function `sub1`. It is normalizable iff (`sub1` M) is normalizable for normalizable M. Since M is in $\mathbf{Norm}_o$, that means, $M \longrightarrow^* \Omega$, or $M \longrightarrow^* n$, or $M \longrightarrow^* \mathsf{error}_1$, or $M \longrightarrow^* \mathsf{error}_2$. In all these cases ($\mathsf{sub1}$ M) is normalizable as the evaluation terminates. The evaluation reduces M first, as indicated by the evaluation context.

Consider the function $\mathsf{is\text{-}const?}$. It is normalizable iff ($\mathsf{is\text{-}const?}$ $(\lambda x.M)$) is normalizable for normalizable $(\lambda x.M)$. Without loss of generality we can consider $(\lambda x.M)$ be a closed phrase.

Consider $M[x/error_1]$ and $M[x/error_2]$. Both these terms must terminate in evaluation, by IH. If they produce error_1 and error_2 respectively, that means, there is a step in the evaluation ($\mathsf{is\text{-}const?}$ $(\lambda x.E[x])$). Therefore, by rule (strict), we can reduce it to 1. If both the terms produce a normal form, then one of the rules (const),(error_1),(error_2) apply. Using an appropriate rule, we can reduce the whole expression ($\mathsf{is\text{-}const?}$ $(\lambda x.M)$) to 0 or error_i. In either case, all these evaluations terminate.

For the rest of the functions, the analysis is similar.

Application: Consider the phrase $(M\ N)$. If $(M\ N)$ is a closed phrase, so must be M and N. By IH, M and N are normalizable; hence $(M\ N)$ is normalizable.

If M and N are open phrases, then $(M\ N)$ is also an open phrase. Any instantiation of $(M\ N)$ is normalizable, since an instantiation $(M'\ N')$ closes the phrases M and N. Therefore, $(M'\ N')$ applies a normalizable term to another normalizable term, and hence is normalizable.

Lambda Abstraction: Consider the phrase $M = (\lambda x.N)$. By IH, N is normalizable. For M to be normalizable $(M\ P)$ must be normalizable for a normalizable P. $(M\ P)$ reduces to $N[x/P]$. Since N is normalizable, all the instantiations of N are normalizable, hence $N[x/P]$ is normalizable, whether it is open or closed.

Therefore, all the terms are normalizable. Hence, all programs must terminate. These programs contain no Y_i's, because we are considering only e_0. $\square$

Now, we can show that e_i terminate for all i.

Lemma 7.7 For any program e, the evaluation of e_i always terminates for any i.

Proof. The proof proceeds by induction on i.

Induction Hypothesis: The evaluation of e_i always terminates under $eval_\Omega$.

Base Case: e_0 always terminates by Lemma 7.6.

Induction Step: Consider e_k. We can form a new term e' by expanding Y_k's into Y_{k-1}'s in e_k. By IH, the evaluation of e' must terminate. It is easy to show that the evaluation of e_k is equivalent to the evaluation of e' by considering the two cases, whether Y_k is reduced or not.

Therefore, the evaluation of e_i terminates for all i. □

Theorem 7.4 The evaluation of a program e terminates with an answer c iff $\mathcal{M}[\![e]\!] = \mathcal{C}[\![c]\!]$.

Proof. ($\Rightarrow$): Let $\mathcal{M}[\![e]\!] = \mathcal{C}[\![c]\!]$. Since $c \neq \Omega$, by Lemma 7.5, there must be e_i such that $\mathcal{M}[\![e]\!] = \mathcal{M}[\![e_i]\!]$. Since the evaluation of e_i always terminates by Lemma 7.7, and since it must be equal to $\mathcal{M}[\![e_i]\!]$ by Lemma 7.4, $eval_\Omega(e_i) = c$. By Lemma 7.3, $eval(e)$ must be equal to c.

($\Leftarrow$): Let $eval(e) = c$. Therefore, by Lemma 7.3, there must be an e_i for which $eval_\Omega(e_i) = c$. From Lemma 7.4, $\mathcal{M}[\![e_i]\!] = \mathcal{C}[\![c]\!]$. Since c is not Ω, $\mathcal{M}[\![e]\!] = \mathcal{M}[\![e_i]\!]$ by Lemma 7.5. Hence, we conclude $\mathcal{M}[\![e]\!] = \mathcal{C}[\![c]\!]$. □

In contrast to adequacy, full abstraction is not easy to prove. In languages such as PCF, the full abstraction result requires addition of a parallel conditional. The full abstraction result in PCF proceeds by showing that all *finite* elements of the model can be represented in the language. If two terms are observationally equivalent, but denotationally different, then there must be finite elements in the domain whose application differentiates the terms. However, since these finite elements are representable in PCF, the same behavior can be mimicked in the operational semantics.

Hence, if the denotations of the terms is different, then they must be observationally be different.

In SPCF, there is a more comprehensive proof for full abstraction. We prove that SPCF is *computationally complete* for its denotational model. From this proof, full abstraction can be obtained trivially.

7.2 The Computational Power of SPCF

Given a model $\mathcal{M}$ for a language L, it is natural to ask whether the language L can express all the *computable* elements in the model. More precisely, the question is whether every recursively enumerable element $d \in \mathcal{T}[\![\tau]\!]$ is definable by a term M of type τ in $\mathcal{L}$, *i.e.*, $\mathcal{M}[\![M]\!] = d$. When this property holds, we say that L is *computationally complete* for $\mathcal{M}$. Plotkin [Plo77] proved that PCF is not computationally complete for the familiar continuous function model. The continuous function model contains "parallel deterministic" functions that PCF *cannot* express. Two well-known, independent examples are the parallel conditional function (pif0) and the parallel *exists* function ($\exists$). The pif0 function is similar to if0, but it returns the least upper bound of the two alternatives if the test argument diverges. The function $\exists$ tests whether its argument, a function from $\mathbf{N}_\perp$ to itself, produces tt (0 in our simplified version of PCF) for any input or whether it always returns ff.

In contrast to PCF and PCF+pif0, SPCF does not need any extensions to be computationally complete. For any computable element in its natural model, there is a term that denotes this element. Given this fact, it is easy to show that the natural model of SPCF is fully abstract.

This section presents a proof of the computational completeness of SPCF. The first subsection defines a generalization of Turing computability suitable for OS-domains. The second subsection applies this definition to SPCF and presents an algorithm for constructing an SPCF term that denotes a given domain element definable by a partial recursive function. This algorithm shows that SPCF is computationally complete for its OS-domains model.

7.2.1 Computability in Observably Sequential Domains

Computability over Scott domains is a generalization of the familiar notion of Turing computability over natural numbers. A computation over a Scott domain produces a recursively enumerable set of finite approximations whose least upper bound is the "answer" produced by the computation. In a computation over a Scott domain D, every finite element is represented by a distinct natural number. The specific choice of representation is determined by the *presentation* of the domain [Sco76]. The presentation assigns a distinct integer code to each finite element and specifies the ordering relationship between any two finite elements.

Domain Presentations

When a Scott domain is an OS-domain, every finite approximation is the least upper bound of its prime approximations. Consequently, a *presentation* only needs to assign integer codes to the prime elements of an OS-domain. Given such a presentation, it is easy to generate unique codes for all finite elements using a standard pairing function.

Note: Since we are not seeking to embed the OS-domains in a universal domain, this presentation is different from the presentation of OS-domains given in Chapter 5. For the computability purposes we rely on the following alternative presentation. This presentation includes an additional predicate that identifies the error elements.

We formalize the concept of a presentation of an OS-domain as follows:

Definition 7.5. (OS-*domain Presentation*) A *presentation* of an OS-domain D with prime-basis $(P, \leq, C)$ is a quadruple of functions $\langle \mathbf{E}, \mathbf{c}, \mathbf{pr}, \mathbf{err} \rangle$:

- a one-to-one function $\mathbf{E} : P \rightarrow \mathbf{N}$, such that $p \leq_P q \Rightarrow \mathbf{E}(p) \leq_{\mathbf{N}} \mathbf{E}(q)$ that maps the elements of P into their codes in the index set in $\mathbf{N}$. By convention, we extend $\mathbf{E}$ to $\perp$ by defining $\mathbf{E}(\perp) \stackrel{df}{=} 0$. Let $\mathcal{I}$ be the range of the function $\mathbf{E}$.

- a function $\mathbf{c} : \mathcal{I} \rightarrow \mathbf{N}$ such that $\forall i, j > 0 : \mathbf{c}(i) = \mathbf{c}(j)$ iff $p_i, p_j \in Q$ for $Q \in C$; by convention, we define $\mathbf{c}(0) \stackrel{df}{=} 0$. Let the range of the function $\mathbf{c}$ be $\mathcal{S}$.

- a predecessor function $\mathbf{pr} : \mathcal{I} \rightarrow \mathcal{I}$ such that $\mathbf{pr}(i) = j$ iff $p_j \prec p_i$. If there is no predecessor for an element p_i, we define $\mathbf{pr}(i) \stackrel{df}{=} 0$. Similarly, we define $\mathbf{pr}(0) \stackrel{df}{=} 0$.

- a function $\mathbf{err} : \mathcal{I} \rightarrow \{0, 1, 2\}$ such that

$$\mathbf{err}(k) = \begin{cases} i & p_k = \mathsf{error}_i^Q \\ 0 & \text{otherwise}. \end{cases}$$

The presentation is *effective* iff the functions $\mathbf{c}$, $\mathbf{pr}$, $\mathbf{err}$ are partial recursive functions mapping $\mathbf{N}$ into $\mathbf{N}$. The function $\mathbf{E}$ is called a *prime enumerator* for D. ∎

The prime enumerator $\mathbf{E}$ maps each prime element to a unique index in $\mathbf{N}$. Hence, $\mathbf{E}$ determines an enumeration of prime elements $p_{i_1}, p_{i_2}, \ldots$ where $\mathbf{E}(p_{i_j}) = i_j$. For the sake of convenience, we require that the indexing determined by $\mathbf{E}$ be topologically sorted.

The remaining three components of a presentation—$\mathbf{c}$, $\mathbf{pr}$, and $\mathbf{err}$—specify how prime elements are partitioned in C-sets, what prime element precedes each C-set, and which prime elements are errors. The function $\mathbf{c}$ implicitly assigns a unique code number in $\mathcal{S} \subseteq \mathbf{N}$ to each C-set in the prime-basis. We refer to the C-set with index m as C_m. As usual, we remove the ambiguity by superscripting the C-sets with the domain whenever needed. This indexing scheme for C-sets determines the following partial order on $\mathcal{S}$, which we will use later in this section.

Definition 7.6. ($\leq'$) Let $m, m' \in \mathbf{N}$ be in the range of the function $\mathbf{c}$. Then $m' \leq' m$ iff for all $q \in C_m$ there exists $p \in C_{m'}$ such that $p \leq q$. ∎

Finite elements are encoded as a finite set of primes as described in Chap. 5. Since the encoding is unique for every n-tuple of positive numbers $\langle k_1, k_2, \ldots, k_n \rangle$, each finite element has a unique encoding. Since the pairing function, the projection functions, and the sorting function over the encoded tuples are recursive, they can be expressed in SPCF as $\langle \cdot, \cdot \rangle$, 1st, 2nd, and sort respectively.

Computable Elements

We have now finally developed all of the machinery required to define the *computable* elements of an OS-domain. Informally, an element d of an effectively presented OS-domain D is computable iff there exists a partial recursive function that "represents" the element. The following definition makes this notion precise.

Definition 7.7. (*Computable Element (OS Domains)*) Let D be an OS-domain with the effective presentation $\langle \mathbf{E}, \mathbf{c}, \mathbf{pr}, \mathbf{err} \rangle$ where $\mathcal{I}$ denotes the range of $\mathbf{E}$ and $\mathcal{S}$ denotes the range of $\mathbf{c}$. An element d of an OS-domain D is *computable* iff the function $f_d : \mathcal{S} \rightarrow_p \mathcal{I}$ defined by

$$f_d(m) = \begin{cases} k & \text{if } \mathbf{c}(k) = m \text{ and } p_k \sqsubseteq d \\ \text{undefined} & \text{otherwise} \end{cases}$$

is a partial recursive function over $\mathbf{N}$. We call the function f_d the *characteristic function* of d. ∎

A Comparison with Classical Domain Theory

The standard definition of computability for Scott domains is similar to our definition of computability for OS-domains. The only difference is that our definition exploits the additional topological structure of OS-domains. It is easy to show that an element d of an OS-domain D with effective presentation $\mathbf{E}$ is computable iff it is computable in the isomorphic Scott domain with the corresponding effective presentation.

Recall Definition 5.3 for Scott presentation of the domain. According to that definition, a presentation δ_s consists of an enumeration $\mathbf{E}_s$ that enumerates all the finite elements and a binary predicate *Con* that checks for consistency and a ternary predicate *Lub* that checks the relation $\sqcup$. Given an effective presentation δ for an *OS*-domain D, it is easy to generate the effective presentation δ_s for D viewed as a Scott domain.

We will show that d is a computable element an *OS*-domain D with effective presentation δ iff d is a computable element in the isomorphic Scott domain with effective presentation δ_s.

Definition 7.8. (*Computable Element (Scott Domains)*) Let D be a Scott domain with the enumerator $\mathbf{E}_s$, and let $\mathcal{I} = range(\mathbf{E}_s)$. An element d of the domain D is computable iff there is a partial recursive predicate g_d over $\mathcal{I}$ such that

$$g_d(i) = \begin{cases} \mathsf{tt} & \text{iff } a_i \sqsubseteq d \\ \text{undefined} & \text{otherwise} \end{cases}$$

We call g_d the (*Scott*) *characteristic function* for d. ∎

In an *OS*-domain D, the two definitions of computability coincide.

Lemma 7.9 Let D be an *OS*-domain with effective presentation δ and let δ_s be the effective Scott for the isomorphic Scott domain D_s. An element $d_s \in D_s$ has a *Scott-characteristic function* g_d iff the corresponding element in $d \in D$ has a characteristic function f_d.

Proof. ($\Leftarrow$): Given f_d we can define the corresponding Scott-characteristic function g_d as follows. The function g_d is defined on *finite elements* a_i by the rule

$$g_d(i) = \mathsf{tt} \iff \forall j : p_j \sqsubseteq a_i : [f_d(\mathbf{c}(j)) = j] .$$

where the index i is the encoding of the tuple $\langle k_1, \ldots, k_n \rangle$. The set of prime indices $\{j \mid p_j \sqsubseteq a_k\}$ is simply $\{k_1, k_2, \ldots, k_n\}$ where $\langle k_1, \langle k_2, \ldots, \langle k_n, 0 \rangle \ldots \rangle \rangle = i$. Hence, $g_d(i)$ is computable for i, an index of a finite element.

(*Rightarrow*): Conversely, given g_d, we can define f_d as follows. Given any prime index $k_1 \in \mathcal{I}$, let $a_{\langle k_1, \ldots, k_n \rangle} = p_{k_1}$ and let $m = \mathbf{c}(k_1)$, the code of the C-set containing

p_{k_1}. Then,

$$f_d(m) = k_1 \iff g_d(\langle k_1, \ldots, k_n \rangle) = \mathsf{tt}.$$

Since only one element of the C-set with index m can approximate d, $f_d(m)$ can be calculated by finding element a_k in C-set C_m such that $g_d(k)$ is true. If such an element exists, a dovetailing search will eventually find it. Then $f_d(m)$ is equal to that element. If such an element does not exist, the computation diverges. □

7.2.2 Computability in SPCF

To apply the preceding general framework to SPCF, we need to define an effective presentation δ^τ for each of the domains $\mathcal{T}[\![t]\!]$ of the SPCF model. We proceed by induction on the structure of the type t. Most of it has been presented in Chapter 5; however, we will repeat it here as the presentation is slightly different.

Conventions. In this subsection, the letters I, J range over indices of *finite* elements, i, j, k, l over indices of prime elements, and m, n over the indices of C-sets. A C-set in the domain $\mathcal{T}[\![\tau]\!]$ is called a c^τ-set. ∎

The design of an enumerator $\mathbf{E}^o : \mathbf{N}_\perp^E \to \mathbf{N}$ for the base domain is straightforward:

$$\begin{aligned}
\mathbf{E}^o(\perp) &= 0 \\
\mathbf{E}^o(\mathsf{error}_1) &= 1 \\
\mathbf{E}^o(\mathsf{error}_2) &= 2 \\
\mathbf{E}^o(x) &= x + 3 \qquad (x \in \mathbf{N})
\end{aligned}$$

It is easy to confirm that the enumerator $\mathbf{E}^o$ is topologically sorted. Moreover, it is effective because the functions $\mathbf{c}^o$, $\mathbf{pr}^o$, and $\mathbf{err}^o$ are definable as primitive recursive functions:

$$\begin{aligned}
\mathbf{c}^o(0) &= 0 \\
\mathbf{c}^o(x+1) &= 1 \\
\mathbf{pr}^o(x) &= 0
\end{aligned}
\qquad
\mathbf{err}^o(x) = \begin{cases} x & \text{if } x = 1 \text{ or } x = 2 \\ 0 & \text{otherwise} \end{cases}$$

The definition of the enumerator $\mathbf{E}^{\sigma\to\tau} : \mathcal{T}[\![\sigma \to \tau]\!] \to \mathbf{N}$ and corresponding functions $\mathbf{c}^{\sigma\to\tau}$, $\mathbf{pr}^{\sigma\to\tau}$, and $\mathbf{err}^{\sigma\to\tau}$ for higher types $\sigma \to \tau$ proceeds by sub-induction on the structure of prime functions using the prime approximation ordering. First, set $\mathbf{E}^{\sigma\to\tau}(\perp_{\sigma\to\tau}) = 0$. Next, let f be a prime function with predecessor f', *i.e.*, $f' \prec f$, and let l be the index of f'. Then,

$$\mathbf{E}^{\sigma\to\tau}(f) = \begin{cases} \langle\langle \mathsf{O}, I, k\rangle, l\rangle & \text{if } f \text{ is an output prime, determined by } f',\ a_I, \text{ and } p_k \\ \langle\langle \mathsf{S}, \langle I, m\rangle, n\rangle, l\rangle & \text{if } f \text{ is a schedule prime with } si_f(a_I, C_m) = C_n \end{cases}$$

where the tags O (for output prime) and S (for schedule prime) stand for 0 and 1, respectively. The functions $\mathbf{c}^{\sigma\to\tau}$, $\mathbf{pr}^{\sigma\to\tau}$ and $\mathbf{err}^{\sigma\to\tau}$ are defined as follows:

$$\begin{array}{rcl} \mathbf{c}^{\sigma\to\tau}(\langle\langle \mathsf{O}, I, k\rangle, l\rangle) & = & \langle\langle I, \mathbf{c}^{\tau}(k)\rangle, l\rangle \\ \mathbf{c}^{\sigma\to\tau}(\langle\langle \mathsf{S}, \langle I, m\rangle, n\rangle, l\rangle) & = & \langle\langle I, m\rangle, l\rangle \\ \mathbf{pr}^{\sigma\to\tau}(\langle\langle \cdots\rangle, l\rangle) & = & l \end{array} \qquad \begin{array}{rcl} \mathbf{err}^{\sigma\to\tau}(\langle\langle \mathsf{O}, I, k\rangle, l\rangle) & = & \mathbf{err}^{\tau}(k) \\ \mathbf{err}^{\sigma\to\tau}(\langle\langle \mathsf{S}, \langle I, m\rangle, n\rangle, l\rangle) & = & 0 \end{array}$$

Since $\langle 0, 0\rangle = 0$, $\mathbf{c}(0) = 0$ and $\mathbf{pr}(0) = 0$ as required. It is straightforward to verify that these functions are legitimate, *i.e.*, follow the conditions of an effective presentation.

Given the preceding definition of computability in the SPCF model, we can now state and prove the central claim of this section: *SPCF is a computationally complete language for its observably sequential model.*

Theorem 7.10 Let τ be an SPCF type. For every computable element $d \in \mathcal{T}[\![\tau]\!]$ there exists a closed SPCF term M such that $\mathcal{M}[\![M]\!] = d$.

Proof Sketch. Let d be a computable element with characteristic function f_d. Since SPCF is a conservative (upward-compatible) extension of PCF, which can express all recursive functions in $\mathbf{N} \to_p \mathbf{N}$, SPCF contains a term N_d that computes the function in $f_d^{\dagger}$ corresponding to f_d. More precisely, $f_d^{\dagger} : \mathbf{N}_{\perp}^{E} \to_{os} \mathbf{N}_{\perp}^{E}$ satisfies the conditions:

$$\begin{array}{rcl} f_d^{\dagger}(x) & = & x \ \text{ iff } x \in \mathbf{E}_{\perp} \\ f_d^{\dagger}(m) & = & n \ \text{ iff } f_d(m) = n \\ f_d^{\dagger}(m) & = & \perp \text{ if } f_d(m) \text{ is undefined.} \end{array}$$

We call $f_d^\dagger$ the SPCF-characteristic function for d.

By the same argument, the functions **c**, **pr**, and **err** from any effective domain presentation have SPCF counterparts **C**, **Pr**, and **Err** corresponding to **c**, **pr**, and **err**.

To prove the theorem, all we have to do is construct an SPCF function Decode^τ that maps $f_d^\dagger$ to the corresponding element d. Unfortunately, the obvious approach to writing Decode^τ fails for higher types τ because it requires writing an inverse function Encode^τ, which maps an element d to the corresponding SPCF characteristic function $f_d^\dagger$. The latter function cannot be written in SPCF because *it is not observably sequential*: It recognizes and discards rather than propagates error elements.

We can circumvent this problem by introducing an alternate functional representation for elements of D. Given an element $d \in D$, the *OS-characteristic* function f_d^* is defined in terms of the the characteristic function f_d for d as follows:

$$
\begin{array}{rcll}
f_d^*(x) & = & x & \text{for } x \in \mathbf{E}_\perp \\
f_d^*(m) & = & k & \text{iff } f_d(m) = k \text{ and } \mathbf{err}(k) = 0 \\
f_d^*(m) & = & \mathsf{error}_i & \text{iff } \exists\, m' \leq' m : \mathbf{err}(f_d(m')) = i \text{ for } i > 0 \\
f_d^*(m) & = & \perp & \text{otherwise.}
\end{array}
$$

Recall that $\leq'$ is defined on the C-set numbers: If there is a prime element in m approximating another prime element in m', then $m \leq' m'$.

Figure 7.4 contains the code of a procedure $\mathsf{convert}$ that converts a characteristic function $f_d^\dagger$ to a *OS*-characteristic function f_d^*, *i.e.*, ($\mathsf{convert}$ N_d) denotes f_d^*. Given an index m for a C-set, ($\mathsf{convert}$ f), the code for the *OS*-characteristic function, works as follows. It visits all C-sets that approximate m, starting at the lowest possible one, and checks whether f maps the current C-set index to an error-generating prime. If so, it generates the correct error value; if not, it outputs the prime index (f m). The correctness of the function depends on correctness of the auxiliary function *NextCset* that gives unique n for a given m' and m such that $m' \prec' n \leq' m$. The correctness of the function *NextCset* can easily be verified by studying the encoding of the C-sets.

$$
\begin{array}{lll}
\mathsf{convert}^{\tau} & = \lambda f.\lambda m. & \\
& \quad \mathbf{letrec}\ z{=}\lambda m'.\ \mathbf{let}\ v{=}(f\ m') & \\
& \qquad\qquad \mathbf{in}\ \ (\mathsf{if0}\ \ (generr^{\tau}\ v) & \\
& \qquad\qquad\qquad (\mathsf{if0}\ \ (=\ m'\ m)\ v & \\
& \qquad\qquad\qquad\qquad (z\ (NextCset\ m'\ m))) & \\
& \qquad\qquad\qquad \Omega) & \\
& \quad \mathbf{in}\ \ (z\ (InitCset^{\tau}\ m)) &
\end{array}
$$

$$
\begin{array}{ll}
generr^{\tau} & = \lambda v.\ \mathbf{let}\ errnum{=}(\mathbf{Err}^{\tau}\ v) \\
& \quad \mathbf{in}\ \ (\mathsf{if0}\ \ (\mathsf{sub1}\ errnum)\ \mathsf{error}_1\ (\mathsf{if0}\ (\mathsf{sub1}^2\ errnum)\ \mathsf{error}_2\ \ulcorner 0 \urcorner))
\end{array}
$$

Comment The function $(NextCset\ m'\ m)$ calculates the unique n such that $m' \prec' n \leq' m$.

$$
\begin{array}{l}
NextCset^{\tau} = \lambda m', m.(\mathsf{if0}\ \ (=\ \ (\mathbf{C}^{\tau}\ (\mathsf{2nd}\ m))\ m') \\
\qquad\qquad\qquad\qquad\quad m \\
\qquad\qquad\qquad\qquad\quad (NextCset^{\tau}\ m'\ (\mathbf{C}^{\tau}\ (\mathsf{2nd}\ m))))
\end{array}
$$

$$InitCset^{\tau} = (NextCset^{\tau}\ \ulcorner 0 \urcorner)$$

Figure 7.4 The Function convert

In contrast to Encode^{τ}, the function encode^{τ} mapping an element $d \in D$ to f_d^* is definable in SPCF. Similarly, the function decode^{τ} mapping a *OS*-characteristic function f_d^* to the corresponding element $d \in D$ is definable in SPCF. If we construct a procedure decode^{τ}, the proof of the theorem is complete, because

$$\mathcal{M}[\![(\mathsf{decode}^{\tau}\ (\mathsf{convert}\ x))]\!] = \mathcal{M}[\![(\mathsf{Decode}^{\tau}\ x)]\!]\,.$$

The function decode relies on the function encode; both these functions are simultaneously constructed inductively. The construction of the code for decode^{τ} and encode^{τ} proceeds by induction on the type τ. At each stage in the construction, we will assume the following induction hypothesis for smaller types, which simply states that encode and decode satisfy their specifications.

Induction Hypothesis: For every type τ:

1. There is an SPCF function decode^τ, which decodes a OS-characteristic function f^*_d into the data object d, *i.e.*,

$$\mathcal{M}[\![\mathsf{decode}^\tau]\!](f^*_d) = d\,.$$

2. There is an SPCF function encode^τ, which encodes a data object $e \in \mathcal{T}[\![\tau]\!]$ as its OS-characteristic function f^*_e, *i.e.*,

$$\mathcal{M}[\![\mathsf{encode}^\tau]\!](e) = f^*_e\,.$$

Consequently, we must show that the induction hypothesis holds for the constructed functions decode^τ and encode^τ.

The definition of the functions decode^o and encode^o for the base type o is straightforward:

$$\begin{array}{lcl} \mathsf{decode}^o & = & \lambda f^*_d.(f^*_d\ 1) \\ \mathsf{encode}^o & = & \lambda x.\lambda n.(\mathsf{if0}\ n\ \ulcorner 0 \urcorner\ (\mathsf{if0}\ (\mathsf{sub1}\ n)\ (+\ \ulcorner 3 \urcorner\ x)\ \Omega)) \end{array}$$

The induction hypothesis obviously holds for type o.

Assume that the induction hypothesis holds for the types σ and τ, implying that we can use the functions decode^σ, encode^σ, decode^τ, and encode^τ in the construction of the functions $\mathsf{decode}^{\sigma\to\tau}$ and $\mathsf{encode}^{\sigma\to\tau}$. The function $\mathsf{decode}^{\sigma\to\tau}$ is shown in Figure 7.5. It accepts an OS-characteristic function f^*_d and some argument e for the function d. It relies on the function encode^σ to encode e as its OS-characteristic function f^*_e. In the body of $\mathsf{decode}^{\sigma\to\tau}$, the auxiliary function *Apply* takes the arguments f^*_d and f^*_e constructs the OS-characteristic function $f^*_{d(e)}$ for $d(e)$. Then $\mathsf{decode}^{\sigma\to\tau}$ decodes the OS-characteristic function $f^*_{d(e)}$ to produce the data object $d(e)$.

To prove that $\mathsf{decode}^{\sigma\to\tau}$ satisfies the induction hypothesis, we observe that by the extensionality of $\mathcal{T}[\![\sigma \to \tau]\!]$

$$\mathcal{M}[\![\mathsf{decode}^{\sigma\to\tau}]\!](f^*_d) = d$$

$$
\begin{array}{lll}
\mathsf{decode}^{\sigma\to\tau} & = \lambda f_d^*.\lambda e^\sigma.(\mathsf{decode}^\tau\ (\mathit{Apply}^\tau\ f_d^*\ (\mathsf{encode}^\sigma\ e))) \\[1em]
\mathit{Apply}^\tau & = \lambda f_d^*, f_e^*.\lambda m\,. \\
& \quad \mathbf{letrec}\ z = \lambda l\,.(\mathbf{let}\ \mathit{final} = (\mathsf{1st}\ l) \\
& \qquad\qquad\qquad \mathbf{in}\ \ (\mathsf{if0}\ \ (=\ O\ (\mathsf{1st}\ \mathit{final})) \\
& \qquad\qquad\qquad\qquad (\mathbf{let}\ \langle I, k\rangle = (\mathsf{2nd}\ \mathit{final}) \\
& \qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ (\mathsf{if0}\ \ (=\ m\ (\mathbf{C}^\tau\ k)) \\
& \qquad\qquad\qquad\qquad\qquad\qquad k \\
& \qquad\qquad\qquad\qquad\qquad\qquad (\mathbf{let*}\ n_\tau = (\mathit{NextCset}^\tau\ (\mathbf{C}^\tau\ k)\ m) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad l = (f_d^*\ \langle\langle I, n\rangle, l\rangle) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ (z\ l)))) \\
& \qquad\qquad\qquad\qquad (\mathbf{let*}\ \langle\langle I, m'\rangle, n\rangle = (\mathsf{2nd}\ \mathit{final}) \\
& \qquad\qquad\qquad\qquad\qquad\ I' = (\mathsf{sort}\ \langle (f_e^*\ n), I\rangle) \\
& \qquad\qquad\qquad\qquad\qquad\ l = (f_d^*\ \langle\langle I', m'\rangle, l\rangle) \\
& \qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ (z\ l)))) \\
& \quad\ \ \mathbf{in}\ \ (z\ \ulcorner 0 \urcorner)
\end{array}
$$

Figure 7.5 The Function *decode*

iff for all finite elements $e \in \mathcal{T}[\![\sigma]\!]$,

$$(\mathcal{M}[\![\mathsf{decode}^{\sigma\to\tau}]\!](f_d^*))(e) = d(e).$$

By the definition of $\mathsf{decode}^{\sigma\to\tau}$, the preceding line reduces to showing

$$\mathcal{M}[\![(\mathsf{decode}^\tau(\mathit{Apply}\ F_d\ (\mathsf{encode}^\sigma\ E)))]\!] = d(e)$$

where the terms F_d and E have the denotations f_d^* and e respectively. The term E exists by the induction hypothesis. If we assume $\mathcal{M}[\![\mathit{Apply}]\!](f_d^*)(f_e^*) = f_{d(e)}^*$ for all d and e, then the proof obligation reduces to

$$\mathcal{M}[\![\mathsf{decode}^\tau]\!](f_{d(e)}^*) = d(e),$$

which holds by the induction hypothesis. Thus, we have reduced the claim about the correctness of decode function to a specific lemma about the function *Apply*.

$$
\begin{array}{lll}
\mathsf{encode}^{\sigma\to\tau} & = & \lambda g.\lambda m. \\
& & \quad \mathbf{letrec}\ z{=}\lambda m'.(\mathsf{if0}\ \ (=\ \ m'\ m)\ (chk^{\sigma\to\tau}\ g\ m) \\
& & \qquad\qquad\qquad\qquad (\mathsf{if0}\ (<?^{\tau}\ (chk^{\sigma\to\tau}\ g\ m')\ m) \\
& & \qquad\qquad\qquad\qquad\quad (z\ (NextCset^{\sigma\to\tau}\ m'\ m)) \\
& & \qquad\qquad\qquad\qquad\quad \Omega)) \\
& & \quad\ \mathbf{in}\ \ (z\ (InitCset\ m))
\end{array}
$$

$$
\begin{array}{lll}
chk^{\sigma\to\tau} & = & \lambda g, m.\ (\mathbf{let}\ \ \langle\langle I, n\rangle, j\rangle =\ \ m \\
& & \quad \mathbf{in}\ \ \mathsf{call/cdc}(\lambda t.(\mathbf{let*}\ \ f^*_{a'} = (Extend^{\sigma}\ I\ n\ j\ t) \\
& & \qquad\qquad\qquad\qquad\quad a' \ = (\mathsf{decode}^{\sigma}\ f^*_{a'}) \\
& & \qquad\qquad\qquad\qquad\quad l \ \ = ((\mathsf{encode}^{\tau}\ (g\ a'))\ n) \\
& & \qquad\qquad\qquad \mathbf{in}\ \ \langle\langle \mathsf{O}, I, l\rangle, j\rangle))))
\end{array}
$$

$$
\begin{array}{lll}
Extend^{\sigma} & = & \lambda I, n, j, t.\lambda m'.(\mathsf{if0}\ \ (InElt^{\sigma}\ I\ m') \\
& & \qquad\qquad\qquad\quad ((GenFun^{\sigma}\ I)\ m') \\
& & \qquad\qquad\qquad\quad (t\ \langle\langle \mathsf{S}, \langle i, n\rangle, (Direction^{\sigma}\ I\ m')\rangle\ j\rangle))
\end{array}
$$

Comment $(GenFun\ I)$ generates $f^*_{a_I}$.

$$
\begin{array}{lll}
GenFun^{\sigma} & = & \lambda I.\,\lambda m.\ (\mathbf{letrec}\ z{=}\lambda I.(\mathsf{if0}\ \ (=\ \ (\mathbf{C}^{\sigma}\ (\mathsf{1st}\ I))\ m) \\
& & \qquad\qquad\qquad\qquad\qquad (\mathsf{1st}\ I) \\
& & \qquad\qquad\qquad\qquad\qquad (z\ (\mathsf{2nd}\ I))) \\
& & \qquad\quad \mathbf{in}\ \ (z\ I))
\end{array}
$$

Figure 7.6 The Function *encode*

Lemma 7.11 Let f^*_d, f^*_e be the *OS*-characteristic functions for $d \in \mathcal{T}[\![\sigma \to \tau]\!]$ and $e \in \mathcal{T}[\![s]\!]$, respectively. Then,

$$\mathcal{M}[\![Apply]\!](f^*_d)(f^*_e) = f^*_{d(e)}.$$

Proof. Set $h = \mathcal{M}[\![Apply]\!](f^*_d)(f^*_e)$. The lemma holds if h is the *OS*-characteristic function for $d(e)$, *i.e.*

$$\forall\, m, i:\ h(m) = i \iff p_i \in d(e) \text{ and } \mathbf{c}^{\tau}(i) = m\,.$$

We show that $h(m)$ gives the correct result by analyzing the three possible cases:

Comment $(\mathit{Direction}\ I\ m) = n$ where a_I is covered by C_n and $n \leq' m$.

$$
\begin{array}{ll}
\mathit{Direction}^\sigma & = \lambda I, m.(\mathsf{if0}\ I \\
& \quad (\mathit{InitCset}\ m) \\
& \quad (\mathsf{if0}\ (<?^\sigma\ (\mathsf{1st}\ I)\ m) \\
& \qquad (\mathit{NextCset}^\sigma\ (\mathbf{C}^\sigma\ (\mathsf{1st}\ I))\ m) \\
& \qquad (\mathit{Direction}^\sigma\ (\mathsf{2nd}\ I)\ m)))
\end{array}
$$

Comment $(\mathit{InElt}\ I\ m) = \ulcorner 0 \urcorner$ iff $p_k \in a_I$ and $\mathbf{c}(k) = m$; otherwise $\ulcorner 1 \urcorner$.

$$
\begin{array}{ll}
\mathit{InElt}^\sigma & = \lambda I, m.(\mathsf{if0}\ \ m\ \ \ulcorner 0 \urcorner \\
& \quad (\mathsf{if0}\ \ I\ \ulcorner 1 \urcorner \\
& \qquad (\mathsf{if0}\ (=\ \ (\mathbf{C}^\sigma\ (\mathsf{1st}\ I))\ m) \\
& \qquad\quad \ulcorner 0 \urcorner \\
& \qquad\quad (\mathit{InElt}^\sigma\ (\mathsf{2nd}\ I)\ m))))
\end{array}
$$

Comment $(<?\ k\ m) = \ulcorner 0 \urcorner$ iff $p_k < C_m$; otherwise $\ulcorner 1 \urcorner$.

$$
\begin{array}{ll}
<?^\tau & = \lambda k, m.(\mathsf{if0}\ m\ \ulcorner 1 \urcorner \\
& \quad (\mathsf{if0}\ (=\ \ k\ (\mathsf{2nd}\ m)) \\
& \qquad \ulcorner 0 \urcorner \\
& \qquad (<?^\tau\ k\ (\mathbf{C}^\tau\ (\mathsf{2nd}\ m)))))
\end{array}
$$

Figure 7.7 Auxiliary Functions

$f^*_{d(e)}(m) = j$. We must show that $h(m) = j$. First, we observe that for any $d \in \mathcal{T}[\![\sigma \to \tau]\!]$, and $e \in \mathcal{T}[\![\sigma]\!]$, $p_j \in d(e)$ iff there exists a least prime $q_* \in d$ such that $q_*(e) = p_j$. Otherwise, d would not have a unique sequentiality index at each point during its exploration of e in the direction $\mathbf{c}^\tau(j)$.

By simple rewriting rules $h(m) = z(0)$ and q_0 approximates q_*. By the following claim, $z(0) = j$, proving the required result for this case.

Claim: $z(l) = j$ if q_l approximates q_*.

We prove the claim by induction on the distance (see Definition 3.10) from q_l to q_*.

Base Case: If the distance is 0 then q_l must be equal to q_*. Then, by simple rewriting rules, the function z generates j.

Inductive Case: The next approximation to q_* is constructed from current approximation l as follows:

- If l is the index of an output prime $\langle\langle \mathsf{O}, I, k\rangle, l'\rangle$, the function d has two options:
  - it can output a prime $p_{k'}$ in the C-set m' immediately above p_k in the direction C^τ_m without further examining the input e; or
  - it can examine the input e above a_I to help determine which prime to output in the C-set m'.

  In either case, the next element above l in d in the direction of q_* must be in the C-set $\langle\langle I, m'\rangle, l\rangle$. Therefore the index of the next approximation to q_* is $f^*_d(\langle\langle I, m'\rangle, l\rangle)$.

- If l is a schedule prime $\langle\langle \mathsf{S}, \langle I, m'\rangle, n\rangle, l'\rangle$, then if $\mathsf{error}_i^{C_n} \in e$ we must have $\mathsf{error}_i^{m'} \in d(e)$. In other words, the function d has to explore the C-set n in the input e where C_n covers a_I. Since q_* exists, this exploration yields a better approximation $a_{I'} = a_I \sqcup r_i$ to the input e where $r_i \in e$ and $\mathbf{c}^\sigma(i) = n$. Therefore, the next prime immediately above l in the direction of q_* (if it exists) is a prime in the C-set $\langle\langle I', m'\rangle, l\rangle$.

Therefore the next approximation decreases the distance from q_* by 1. By the induction hypothesis, z applied to this better approximation gives j as the result.

$f^*_{d(e)}(m) = \mathsf{error}_i$. Let m' be the C-set approximating m and $\mathsf{error}_i^{C_{m'}} \in d(e)$. Then q_* be the maximum prime in d such that $q_*(e) \prec C_{m'}$.

By rewriting the code, $h(m) = z(0)$. Since q_0 approximates q_*, the following claim gives $h(m) = \mathsf{error}_i$.

Claim: If q_l approximates q_* then $z(l) = \mathsf{error}_i$.

The proof of the claim proceeds by induction on the distance from q_l to q_*.

Base Case: If $q_l = q_*$, then the function h has two choices depending on q_l. If it is an output prime, it will generate an error without looking for further input. If it is a schedule prime, it will generate an error while looking at an error in the input e. In either case the corresponding error value error_i is generated.

Inductive Case: As in the preceding case, each time z reduces the distance between q_* and q_l by 1. Therefore, by the induction hypothesis, $z(l) = \mathsf{error}_i$.

$f^*_{d(e)}(m) = \bot$. There are two possibilities:

- $\exists p_{j'} \in d(e) \cap C^\tau_{m'}$ such that $m' \leq' m$ and $p_{j'}$ is not in the direction C^τ_m. In other words, there is a conflicting prime along the direction C^τ_m.
- $\exists p_{j'} \in d(e)$ such that $p_{j'} \notin C^\tau_m$ and it is the maximum prime in the direction C^τ_m.

In either case, divergence of the function $h(m)$ can be proved by an analysis of the behavior of z as in the preceding cases.

Thus, in each of the above cases, $h(m)$ returns the same result as $f^*_{d(e)}$. That is, h is the OS-characteristic function for $d(e)$. Hence, we proved the claim $\mathcal{M}[\![Apply]\!](f^*_d)(f^*_e)$ $= f^*_{d(e)}$.

The remaining obligation in the proof of Theorem 7.10 is to show that the family of functions $\mathsf{encode}^{\sigma\to\tau}$ satisfy the condition specified in the induction hypothesis. The definition of the functions $\mathsf{encode}^{\sigma\to\tau}$ is given in Figure 7.6.

For the moment, let us assume that the auxiliary function chk satisfies its specification: $chk(g, m) = f^*_g(m)$ if $\exists p_k \in g : p_k \prec C^{\sigma\to\tau}_m$. We can prove that $\mathsf{encode}^{\sigma\to\tau}$ returns f^*_g for a data object $g \in \mathcal{T}[\![\sigma \to \tau]\!]$ as follows.

First, we observe that *InitCset* is an auxiliary function that returns the index of first C-set above 0 that approximates the given input. If the input is 0 it returns 0.

Therefore the result of $InitCset(m)$ always approximates m and there is no non-zero C-set that approximates the $InitCset(m)$.

Set $h = \mathcal{M}[\![\mathsf{encode}]\!](g)$. The function encode satisfies the induction hypothesis if $h = f_g^*$. We prove this equality by analyzing the possible results of $f_d^*(m)$:

$f_g^*(m) \in \mathbf{N}$: Therefore, we must prove that $h(m) = f_g^*(m)$. By simple rewriting rules, $h(m) = z(InitCset(m))$. If we prove the following claim, we have the result that $h(m) = f_g^*(m)$ since $InitCset(m) \leq' m$.

Claim: If $m' \leq' m$, then $z(m') = f_g^*(m)$.

The proof of the claim is by induction on the distance between the C-sets (see Definition 3.10) $C_{m'}$ and C_m.

Base Case: If the distance is 0 then $m' = m$. Since $\exists p_k \in g : p_k \prec C_{m'}$, the function $chk(g, m)$ returns $f_g^*(m)$ by Lemma 7.12. Therefore, $z(m') = f_g^*(m)$.

Inductive Case: By the assumption that $f_g^*(m) \in \mathbf{N}$, the function $chk(g, m')$ returns a prime in the direction C_m. Therefore, z is recursively invoked with $NextCset(m', m)$, the index of the C-set one step closer to m. By induction hypothesis and simple rewriting, $z(m') = z(NextCset(m', m)) = f_g^*(m)$.

$f_g^*(m) = \mathsf{error}_i$: We must prove that $h(m)$ also generates the error element error_i, for $i = 1, 2$. As in the preceding case, $h(m) = z(InitCset(m))$.

Since $f_g^*(m) = \mathsf{error}_i$ there must be a least $n \leq' m$ where $f_g^*(n) = \mathsf{error}_i$. Since $InitCset(m) \leq' n$, we can prove the required result for this case by proving the following claim.

Claim: If $m' \leq' n$, then $z(m') = \mathsf{error}_i$.

Proof of the claim proceeds by induction on the distance between m' and n. It is similar to the preceding case.

$f_g^*(m) = \bot$: We must prove that $h(m) = \bot$. Then only one of the statements follows:

- There is a p_k in g such that and p_k is not in the direction C_m and $p_k \in C_n$ where $n \leq' m$.
- There is a p_k in g such that p_k is the maximum prime in the direction C_m. Let C_n be the C-set covering p_k. It is easy to see $n \leq' m$.

In either case, the claim that $m' \leq' n$ implies $z(m') = \perp$ can be proved by induction on the distance from m' and n. Therefore $h(m) = z(\mathit{InitCset}(m)) = \perp$.

Therefore, $f^*_g(m) = h(m)$ for all valid values of m, implying that encode satisfies its specification. □

Hence, we have reduced the proof of Theorem 7.10 to the proof of the following lemma.

Lemma 7.12 Let $g \in \mathcal{T}[\![\sigma \to \tau]\!]$ and m be the C-set code in the domain $\mathcal{T}[\![\sigma \to \tau]\!]$. If there is a $p_k \in g$ such that p_k is covered by $C^{\sigma\to\tau}_m$, then $chk(g, m) = f^*_g(m)$.

Proof. Let g and m be the inputs to the function chk and let $m = \langle I, n, j \rangle$. Then $p_j \in \mathcal{T}[\![\sigma \to \tau]\!]$ approximates g by assumption. In addition, by the definition of our coding scheme, C_n covers $p_j(a_I)$.

We first observe that chk, with the help of *Extend*, generates a *OS*-characteristic function $f^*_{a'}$. This function agrees with $f^*_{a_I}$ on the queries m' such that $\mathbf{c}^\sigma(j) = m$ and $p_j \in a_I$; but, if the function $f^*_{a'}$ is explored beyond a_I, it "throws" an index back to chk, which returns this index as the final result.

The specifications for the auxiliary functions *GenFun*, *Direction*, *InElt* and <? is given in the comments. It is straightforward to verify that the functions follow the specifications by induction on encodings of the primes and C-sets.

Now we analyze the four possible cases of $f^*_g(m)$:

$f^*_g(m) = \perp$: In this case, we must show that $chk(g, m) = \perp$. Since $m = \langle I, n, j \rangle$, we know that $f^*_g(m)$ is $\perp$ iff $g(a_I) \cap C^\tau_n$ is empty. In other words, there is no prime element in the C-set C_n^τ at the output of $g(a_I)$. Therefore, by the induction hypothesis of encode, $\mathsf{encode}^\tau(g(a_I))(n)$ should diverge. Since $a_I = a'$ for all the queries within a_I, the function *chk* diverges on the given inputs g and m.

$f_g^*(m) = \mathsf{error}_i$: Just as in the preceding case, $g(a_I)$ should have an error prime, $\mathsf{error}_i^{C_n}$. By the induction hypothesis of encode, $\mathsf{encode}^\tau(g(a_I))(n)$ returns error_i; therefore the result of the function *chk* is error_i.

$f_g^*(m) = \langle\langle \mathsf{O}, I, l\rangle, j\rangle$: In that case, $g(a_I)$ should have the prime element p_l at the C-set C_n^τ. By the induction hypothesis of encode, $\mathsf{encode}^\tau(g(a_I))(n) = l$. Therefore, the function *chk* returns the index $\langle\langle \mathsf{O}, I, l\rangle, j\rangle$. Notice that by the assumption that $p_k \prec C_m^{\sigma\to\tau}$ is in g, all of the input a_I is necessary for generating the output at the C-set C_n^τ, which means that the index of the output prime is correct.

$f_g^*(m) = \langle\langle \mathsf{S}, \langle I, n\rangle, n'\rangle, j\rangle$: Therefore, $\mathsf{error}_i^{C_n} \in g(a \sqcup \mathsf{error}_i^{C_{n'}})$. That means, the function g checks the input at the C-set $C_{n'}^\sigma$ to generate the output at the C-set C_n^τ. When the function $f_{a'}^*$ is queried for the input at the C-set $C_{n'}^\sigma$ it throws the index $\langle\langle \mathsf{S}, \langle I, n\rangle, n'\rangle, j\rangle$, which is returned by the function *chk*.

Therefore, in all possible cases, the function $chk(g, m)$ returns $f_g^*(m)$ assuming that the prime covered by m is in g. $\square$

Thus, we conclude the proof of theorem 7.10.

As a corollary, we can prove that every finite element of the domain can be represented in SPCF. This corollary is useful in proving full abstraction.

Corollary 7.13 For any type τ, all the finite elements of $\mathcal{T}[\![\tau]\!]$, can be represented in SPCF. That is,

$$\forall\tau : \forall d \in \mathcal{T}[\![\tau]\!] : \exists D^\tau : \mathcal{M}[\![D]\!] = d$$

Proof. It is easy to use the function *Genfun* in generating the appropriate SPCF term, as it can generate the OS-characteristic function for any finite element in the domain $\mathcal{T}[\![\tau]\!]$ specified by the indices of approximating primes. That is,

$$\mathcal{M}[\![(\mathsf{decode}^{\tau}\ (Genfun\ I))]\!] = a_I$$

where I is the sorted list of indices of prime approximations to a_I. $\square$

Now, we prove that the denotational model we provided for SPCF is fully abstract.

Corollary 7.14 The language SPCF is fully abstract for its denotational model, *i.e.*, observational equivalence of two terms implies denotational equivalence of those terms.

Proof. We proceed by induction on types.

Induction Hypothesis: For all phrases M and N of type τ, if M and N are observationally equivalent, $\forall \rho : \mathcal{M}[\![M]\!]\rho = \mathcal{M}[\![N]\!]\rho$.

Base Case: Let M and N be of type o. If they are closed, the observational equivalence would imply $\mathcal{M}[\![M]\!] = \mathcal{M}[\![N]\!]$ trivially.

If M is open in some variables, N must be open in the same variables; otherwise, we can find a context for which $C[M]$ is closed and $C[N]$ is not, falsifying the premise of observational equivalence. Let $x_1, x_2, \ldots, x_n$ be the free variables in M. Let $\rho(x_i) = d_i$. Therefore, by corollary 7.13, there is an SPCF term D_i such that $\mathcal{M}[\![D_i]\!] = d_i$. Then the following equation holds:

$$\mathcal{M}[\![M]\!]\rho = \mathcal{M}[\![((\lambda x_1, x_2, \ldots x_n.M)D_1\ D_2\ \ldots D_n)]\!]$$

By choosing the context $C[\ ] = ((\lambda x_1, \ldots, x_n.[\ ])D_1\ \ldots D_n)$ as the program context, from observational equivalence we infer

$$\mathcal{M}[\![C[M]]\!] = \mathcal{M}[\![C[N]]\!]$$

thereby proving the required result.

Induction Step: Let M and N be two observationally equivalent terms belonging to type $\sigma \to \tau$. Assume they differ denotationally for some ρ. Therefore, they denote two different functions. By order-extensionality of functions, that would mean there is a finite element d in $\mathcal{T}[\![\sigma]\!]$ for which the $\mathcal{M}[\![M]\!]\rho(d) \neq \mathcal{M}[\![N]\!]\rho(d)$. Since d is a finite element, by corollary 7.13, there is an SPCF term D such that $\mathcal{M}[\![D]\!] = d$. Since $\mathcal{M}[\![M]\!]\rho(d) = \mathcal{M}[\![(M\ D)]\!]\rho$, the following equation can be inferred.

$$\mathcal{M}[\![(M\ D)]\!]\rho \neq \mathcal{M}[\![(N\ D)]\!]\rho.$$

However, by IH, since $(M\ D)$ and $(N\ D)$ are observationally equivalent, they must be denotationally equivalent, contradicting the assumption that M and N are denotationally not equivalent. Therefore, M and N must be denotationally equivalent.

The preceding proof establishes the full abstraction result. $\square$

Chapter 8

Conclusion

In this chapter, we summarize the results of the thesis and discuss the directions in which the results of this thesis can be extended. We also discuss some of the related problems that have not been addressed in the thesis.

8.1 Contributions

In this thesis, we presented a universal framework for higher-order sequential computation. This framework is based on a new formulation of domain theory directed at sequential computation.

Earlier research by Cartwright and Demers suggested that the continuous functions are too general to serve as a basis for a comprehensive model of higher order computation. As a result, we focused on the class of *observably sequential functions* (OS-functions) recently introduced by Cartwright and Felleisen. We determined that these functions can be defined over a class of domains called OS-domains. These domains have "error" elements that have natural counterparts in programming languages. We showed that the OS-domains and the OS-functions form a cartesian closed category. Thus these domains are adequate to define the semantics for higher order sequential programming languages based on standard techniques.

To provide a framework for describing sequential computations over arbitrary OS-domains we constructed two new universal domains. In these domains, we can embed all the OS-domains while preserving the termination behavior of computations.

For these universal domains, we defined two new universal languages. These languages are computationally complete for the corresponding universal domains.

Hence, it is possible to perform arbitrary computations over an arbitrary OS-domain, first by embedding it in the universal domain and later computing over the image in the universal domain.

For the first universal language, we presented two different operational semantics: a lazy, demand-driven evaluator and an full (aggressive) evaluator. We showed that these two evaluators are closely related; in fact, either can be derived from the other.

We also studied SPCF, a sequential language based on PCF. We constructed a semantics for SPCF based on OS-functions that is more "abstract" than (yet isomorphic to) previous models given by Cartwright and Felleisen, and Curien. The "abstractness" of our model made it much easier to prove that SPCF is computationally complete. Computational completeness trivially implies full abstraction for SPCF.

8.2 Future Work

There are two interesting directions for building on the results of this thesis.

First, we can construct a more practical universal language based on Pure Scheme. The language KL is patterned after first-order recursion equations. While this makes it easier to study the theoretical results, it also makes it tedious to program in it. We can adapt Pure Scheme [SF91], as a typeless variant of SPCF, for programming in the universal domain. To show that Pure Scheme is universal, all that we need to show is that it is computationally equivalent to KL.

Second, we can apply the insights from this thesis to the problem of designing data specification languages. An OS-domain presentation is mathematically elegant, but pragmatically crude way to specify a domain. We should be able to design a domain specification language with a syntax similar to (but more general than) ML `datatype` definitions from which effective domain presentations can be extracted. We can view such a data definition language as a component of a executable specification language that that is universal for sequential computation. In this context, denotational definitions for programming languages could be written and executed as prototype interpreters, eliminating the need for a separate operational definition.

8.3 Open Problems

This thesis defines a complete framework for sequential computation: a universal domain with algorithms for embedding arbitrary domains and a corresponding language. Indeed, it defines two universal domains based on the two separate sets of requirements for the embedding. However, it does not answer one question: What is the natural class of functions that embeds only and all OS-domains in the universal domain?

In the thesis, we showed that the standard class of retractions or partial closures does not work. They define some domains that are not OS-domains. In such domains, it is unclear what sequential computation means. It may be possible to impose additional constraints on the class of embedding functions such that only OS-domains can be defined.

The second open problem concerns with the relation of OS-functions to the pure "sequential functions". The functions defined by pedagogic λ-calculus-based languages like PCF are sequential rather than observably sequential. At this point, we do not know how to filter out sequential functions from OS-functions. We believe such a filtering will shed insight into the construction of fully abstract models for PCF like languages.

Bibliography

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984. Revised edition.

[BC82] G. Berry and P-L. Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.*, 20:265–321, 1982.

[BC85] G. Berry and P.-L. Curien. Theory and practice of sequential algorithms: the kernel of the applicative language cds. In J. Reynolds and M. Nivat, editors, *Algebraic Methods in Semantics*, pages 35–88. Cambridge University Press, London, 1985.

[Ber79] G. Berry. *Modèles complètement adéquats et stables des lambda-calculus typé.* PhD thesis, Université Paris VII, 1979.

[CCF94] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract models of observably sequential languages. *Information and Computation*, 111, June 1994.

[CD82] R. Cartwright and J. Donahue. The semantics of lazy (and industrious) evaluation. In *Proc. 1982 ACM Symposium on Lisp and Functional Programming*, pages 253–264, 1982.

[CD88] R. Cartwright and A. Demers. The topology of program termination. In *Proc. Symposium on Logic in Computer Science*, pages 296–308, 1988.

[CF91] R.S. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. Technical Report 91-167, Rice Univeristy Department of Computer Science, 1991.

[CF92] Robert Cartwright and Matthias Felleisen. Observable sequentiality and full abstraction. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 328–342, January 1992.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *Americal Journal of Mathematics*, 58:345–363, 1936.

[Cur92] P.-L. Curien. Observable algorithms on concrete data structures. In *Proc 7th Symposium on Logic in Computer Science*, 1992.

[Cur93] P.L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. Birkhauser, 1993.

[Dro89] Manfred Droste. Recursive domain equations for concrete data structures. *Information and Computation*, 82(1):65–80, July 1989.

[Dro91] Manfred Droste. Universal homogeneous event structures and domains. *Information and Computation*, 94(1):48–61, September 1991.

[Fel87] M. Felleisen. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana University, 1987.

[Fel90] M. Felleisen. On the expressive power of programming languages. In N. Jones, editor, *Proc. 1990 European Symposium on Programming*, pages 134–151. Lecture Notes in Computer Science 432, 1990.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages.* The MIT Press, Cambridge, 1992. Foundations of Computing Series.

[JG76] E. Wagner J. Goguen, J.Thatcher. An initial algebra approach to the specification, correctness, and implementation of abstract data types. Technical Report 6487, IBM Research Report, 1976.

[KC93] Ramarao Kanneganti and Robert Cartwright. What is a universal higher-order programming language? In *Proceedings of ICALP 92*, volume 700. Springer Verlag Lecture Notes in Computer Science, 1993.

[KCF93] R. Kanneganti, R. Cartwright, and M. Felleisen. Spcf: its model, calculus, and computational power. In *Proc. REX Workshop on Semantics and Concurrency*, pages 318–347. Lecture Notes in Computer Science 666. Springer Verlag, 1993.

[Kle36] Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.

[Koy84] C. P. J. Koymans. *Models of the lambda calculus.* Centrum voor Wiskunde en Informatica, Amsterdam, 1984.

[KP93] G. Kahn and G. Plotkin. Concrete domains. In *Festschrift for C. Boehm*, 1993. Previously avilable as Technical Report 336, INRIA LABORIA, 1978.

[M$^+$65] J. McCarthy et al. *Lisp 1.5 Programmer's Manual.* The MIT Press, Cambridge, 2nd edition, 1965.

[Mey82] A.R Meyer. What is a model of λ-calculus? *Information and Computation*, 52:87–122, 1982.

[Plo77] G.D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5:223–255, 1977.

[Plo78] G.D. Plotkin. T^ω as a universal domain. *J. Comput. Sci.*, 17:209–236, 1978.

[Rog67] Hartley Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1967.

[Sco76] D.S. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.

[Sco80] D. S. Scott. Relating theories of the lambda-calculus. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory, Lambda-Calculus and Formalism*, pages 401–450. Academic Press, New York and London, 1980.

[Sco81] D. S. Scott. Lectures on a mathematical theory of computation. Technical Report PRG-19, Oxford University Computing Laboratory, 1981.

[Sco82] D.S. Scott. Domains for denotational semantics. ICALP, July 1982.

[SF91] D. Sitaram and M. Felleisen. Modeling continuations without continuations. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 185–196, 1991.

[Sto81] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Languages.* The MIT Press, Cambridge, Mass., 1981.

[Tur37] Alan M. Turing. Computability and λ-definability. *The Jounal of Symbolic Logic*, 2:153–163, 1937.

[Wan88] M. Wand. Final algebraic semantics and data type extensions. *J. Comput. Syst. Sci.*, 19:27–44, 1988.

[Win80] G. Winskel. *Events in computations.* PhD thesis, Univ. of Edinburgh, 1980.