February 1989

Semantic Domains and Denotational Semantics

Carl A. Gunter
University of Pennsylvania

Peter D. Mosses
University of Pennsylvania

Dana S. Scott
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-16.

Semantic Domains and Denotational Semantics

Abstract

The theory of domains was established in order to have appropriate spaces on which to define semantic functions for the denotational approach to programming-language semantics. There were two needs: first, there had to be spaces of several different types available to mirror both the type distinctions in the languages and also to allow for different kinds of semantical constructs - especially in dealing with languages with side effects; and second, the theory had to account for computability properties of functions - if the theory was going to be realistic. The first need is complicated by the fact that types can be both compound (or made up from other types) and recursive (or self-referential), and that a high-level language of types and a suitable semantics of types is required to explain what is going on. The second need is complicated by these complications of the semantical definitions and the fact that it has to be checked that the level of abstraction reached still allows a precise definition of computability.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-16.

SEMANTIC DOMAINS AND DENOTATIONAL SEMANTICS

Carl A. Gunter
Peter D. Mosses
and Dana S. Scott

MS-CIS-89-16
LOGIC & COMPUTATION 04

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

February 1989

To appear in North Holland's Handbook of Theoretical Computer Science.

Acknowledgements: This research was supported in part by DARPA grant NOOO14-85-K-0018, NSF grants MCS-8219196-CER, IRI84-10413-AO2 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

SEMANTIC DOMAINS AND DENOTATIONAL SEMANTICS[1]

Carl A. Gunter Peter D. Mosses Dana S. Scott

February 14, 1989

[1]To appear in North Holland's **Handbook of Theoretical Computer Science.**

Contents

List of Figures

List of Tables

Chapter 1

Semantic Domains.

1.1 Introduction.

The theory of domains was established in order to have appropriate spaces on which to define semantic functions for the denotational approach to programming-language semantics. There were two needs: first, there had to be spaces of several different types available to mirror both the type distinctions in the languages and also to allow for different kinds of semantical constructs—especially in dealing with languages with side effects; and second, the theory had to account for computability properties of functions—if the theory was going to be realistic. The first need is complicated by the fact that types can be both compound (or made up from other types) and recursive (or self-referential), and that a high-level language of types and a suitable semantics of types is required to explain what is going on. The second need is complicated by these complications of the semantical definitions and the fact that it has to be checked that the level of abstraction reached still allows a precise definition of computability.

This degree of abstraction had only partly been served by the state of recursion theory in 1969 when the senior author of this report started working on denotational semantics in collaboration with Christopher Strachey. In order to fix some mathematical precision, he took over some definitions of recursion theorists such as Kleene, Nerode, Davis, and Platek and gave an approach to a simple type theory of higher-type functionals. It was only after giving an abstract characterization of the spaces obtained (through the construction of bases) that he realized that recursive definitions of types could be accommodated as well—and that the recursive definitions could incorporate function spaces as well. Though it was not the original intention to find semantics of the so-called untyped λ-calculus, such a semantics emerged along with many ways of interpreting a very large variety of languages.

A large number of people have made essential contributions to the subsequent developments, and they have shown in particular that domain theory is not one monolithic theory, but that there are several different kinds of constructions giving classes of domains appropriate for different mixtures of constructs. The story is, in fact, far from finished even today. In this report we will only be able to touch on a few of the possibilities, but we give pointers to the literature. Also, we have attempted to explain the foundations in an elementary way—avoiding heavy prerequisites (such as category theory) but still maintaining some level of abstraction—with the hope that such an introduction will aid the reader in going further into the theory.

The chapter is divided into seven sections. In the second section we introduce a simple class of ordered structures and discuss the idea of fixed points of continuous functions as meanings for recursive programs. In the third section we discuss computable functions and effective presentations. The fourth section defines some of the operators and functions which are used in semantic definitions and describes their distinguishing characteristics. A special collection of such operators called *powerdomains* are discussed in the fifth section. Closure problems with respect to the *convex* powerdomain motivate the introduction of the class of *bifinite* domains which we describe in the sixth section. The seventh section deals with the important issue of obtaining fixed points for (certain) operators on *domains*. We illustrate the method by showing how to find domains D

satisfying isomorphisms such as $D \cong D \times D \cong D \to D$ and $D \cong \mathsf{N} + (D \to D)$. (Such domains are models of the above-mentioned untyped λ-calculus.)

Many of the proofs for results presented below are sketched or omitted. With a few exceptions, the enthusiastic reader should be able to fill in proofs without great difficulty. For the exceptions we provide a warning and a pointer to the literature.

1.2 Recursive definitions of functions.

It is the essential purpose of the theory of domains to study classes of spaces which may be used to give semantics for recursive definitions. In this section we discuss spaces having certain kinds of limits in which a useful fixed point existence theorem holds. We will briefly indicate how this theorem can be used in semantic specification.

1.2.1 Cpo's and the Fixed Point Theorem.

A *partially ordered set* is a set D together with a binary relation $\sqsubseteq$ which is reflexive, anti-symmetric and transitive. We will usually write D for the pair $\langle D, \sqsubseteq \rangle$ and abbreviate the phrase "partially ordered set" with the term "poset". A subset $M \subseteq D$ is *directed* if, for every finite set $u \subseteq M$, there is an upper bound $x \in M$ for u. A poset D is *complete* (and hence a *cpo*) if every directed subset $M \subseteq D$ has a least upper bound $\bigsqcup M$ and there is a least element $\bot_D$ in D. When D is understood from context, the subscript on $\bot_D$ will usually be dropped.

It is not hard to see that *any* finite poset that has a least element is a cpo. The easiest such example is the one point poset I. Another easy example which will come up later is the poset O which has two distinct elements $\top$ and $\bot$ with $\bot \sqsubseteq \top$. The *truth value cpo* T is the poset which has three distinct points, $\bot, \mathbf{true}, \mathbf{false}$, where $\bot \sqsubseteq \mathbf{true}$ and $\bot \sqsubseteq \mathbf{false}$ (see Figure 1.1). To get an example of an infinite cpo, consider the set N of natural numbers with the discrete ordering (*i.e.* $n \sqsubseteq m$ if and only if $n = m$). To get a cpo, we need to add a "bottom" element to N. The result is a cpo $\mathsf{N}_\bot$ which is pictured in Figure 1.1. This is a rather simple example because it does not have any interesting directed subsets. Consider the ordinal ω; it is not a cpo because it has a directed subset (namely ω itself) which has no least upper bound. To get a cpo, one needs to add a top element to get the cpo $\omega^\top$ pictured in Figure 1.1. For a more subtle class of examples of cpo's, let $\mathcal{P}S$ be the set of (all) subsets of a set S. Ordered by ordinary set inclusion, $\mathcal{P}S$ forms a cpo whose least upper bound operation is just set inclusion. As a last example, consider the set Q of rational numbers with their usual ordering. Of course, Q lacks the bottom and top elements, but there is another problem which causes Q to fail to be a cpo: Q lacks, for example, the square root of 2! However, the unit interval $[0, 1]$ of real numbers *does* form a cpo.

Given cpo's D and E, a function $f : D \to E$ is monotone if $f(x) \sqsubseteq f(y)$ whenever $x \sqsubseteq y$. If f is monotone and $f(\bigsqcup M) = \bigsqcup f(M)$ for every directed M, then f is said to be *continuous*. A function $f : D \to E$ is said to be *strict* if $f(\bot) = \bot$. We will usually write $f : D \multimap E$ to indicate that f is strict. If $f, g : D \to E$, then we say that $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for every $x \in D$. With this ordering, the poset of continuous functions $D \to E$ is itself a cpo. Similarly, the poset of strict continuous functions $D \multimap E$ is also a cpo. (Warning: we use the notation $f : D \to E$ to indicate that f is a function with domain D and codomain E in the usual set-theoretic sense. On the other hand, $f \in D \to E$ means that $f : D \to E$ is continuous. A similar convention applies to $D \multimap E$.)

To get a few examples of continuous functions, note that when $f : D \to E$ is monotone and D is finite, then f is continuous. In fact, this is true whenever D has no infinite ascending chains.

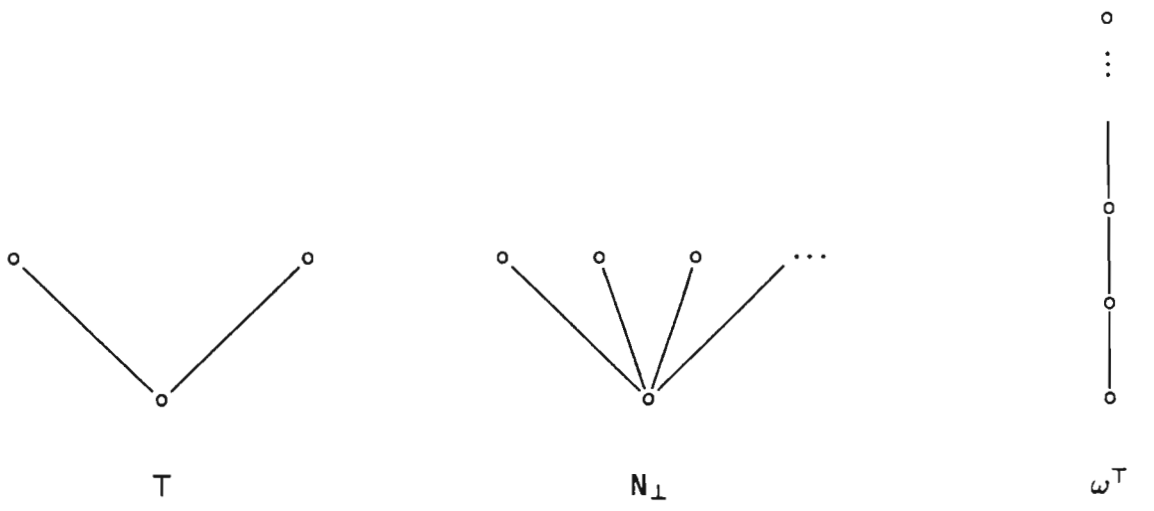


Figure 1.1: Examples of cpo's.

For example, any monotone function $f : \mathbf{N}_\perp \to E$ is continuous. On the other hand, the function $f : \omega^\top \to \mathbf{O}$ which sends the elements of ω to $\perp$ and sends $\top$ to $\top$ is monotone, but it is not continuous. Given sets S, T and function $f : S \to T$ we define the *extension* of f to be the function $f^* : \mathcal{P}S \to \mathcal{P}T$ given by taking

$$f^*(X) = \{f(x) \mid x \in X\}$$

for each subset $X \subseteq S$. The function f^* is monotone and, for any collection X_i of subsets of S, we have

$$f^*(\bigcup_i X_i) = \bigcup_i f^*(X_i).$$

In particular, f^* is continuous. For readers who know a bit about functions on the real numbers, it is worth noting that a function $f : [0,1] \to [0,1]$ on the unit interval may be continuous in the cpo sense without being continuous in the usual sense.

Now, the central theorem may be stated as follows:

Theorem 1 (Fixed Point) *If D is a cpo and $f : D \to D$ is continuous, then there is a point $\mathsf{fix}(f) \in D$ such that $\mathsf{fix}(f) = f(\mathsf{fix}(f))$ and $\mathsf{fix}(f) \sqsubseteq x$ for any $x \in D$ such that $x = f(x)$. In other words, $\mathsf{fix}(f)$ is the least fixed point of f.*

Proof: Note that $\perp \sqsubseteq f(\perp)$. By an induction on n using the monotonicity of f, it is easy to see that $f^n(\perp) \sqsubseteq f^{n+1}(\perp)$ for every n. Set $\mathsf{fix}(f) = \bigsqcup_n f^n(\perp)$. By the continuity of f, it is easy to see that $\mathsf{fix}(f)$ is a fixed point of f. To see that it is the least such, note that if x is a fixed point of f, then, for each n, $f^n(\perp) \sqsubseteq f^n(x) = x$. ∎

1.2.2 Some applications of the Fixed Point Theorem.

The factorial function. As a first illustration of the use of the Fixed Point Theorem, let us consider how one might define the *factorial function* $\mathsf{fact} : \mathsf{N}_\perp \to \mathsf{N}_\perp$. The usual approach is to say that the factorial function is a strict function which satisfies the following recursive equation for each number n:

$$\mathsf{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \mathsf{fact}(n-1) & \text{if } n > 0. \end{cases}$$

where $*, - : \mathsf{N} \times \mathsf{N} \to \mathsf{N}$ are multiplication and subtraction respectively. But how do we know that there *is* a function fact which satisfies this equation? Define a function

$$F : (\mathsf{N}_\perp \multimap \mathsf{N}_\perp) \to (\mathsf{N}_\perp \multimap \mathsf{N}_\perp)$$

by setting:

$$F(f)(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * f(n-1) & \text{if } n > 0 \\ \perp & \text{if } n = \perp \end{cases}$$

for each $f : \mathsf{N}_\perp \multimap \mathsf{N}_\perp$. The definition of F is *not* recursive (F appears only on the left side of the equation) so F certainly exists. Moreover, it is easy to check that F is continuous (but not strict). Hence, by the Fixed Point Theorem, F has a least fixed point $\mathsf{fix}(F)$ and this solution will satisfy the equation for fact.

Context Free Grammars. One familiar kind of recursion equation is a context free grammar. Let Σ be an alphabet. One uses context free grammars to specify subsets of the collection Σ^* of finite sequences of letters from Σ.[1] Here are some easy examples:

1. $$E ::= \epsilon \mid Ea$$

   defines the strings of a's (including the empty string ϵ).

2. $$E ::= a \mid bEb$$

   defines strings consisting either of the letter a alone or a string of n b's followed by an a followed by n more b's.

3. $$E ::= \epsilon \mid aa \mid EE$$

   defines strings of a's of even length.

[1] The superscripted asterisk will be used in three entirely different ways in this chapter. Unfortunately, all of these usages are standard. Fortunately, however, it is usually easy to tell which meaning is correct from context.

We may use the Fixed Point Theorem to provide a precise explanation of the semantics of these grammars. Since the operations $X \mapsto \{\epsilon\} \cup X\{a\}$, $X \mapsto \{a\} \cup \{b\}X\{b\}$, and $X \mapsto \{\epsilon\} \cup \{a\}\{a\} \cup XX$ are all continuous in the variable X, it follows from the Fixed Point Theorem that equations such as

1. $X = \{\epsilon\} \cup X\{a\}$
2. $X = \{a\} \cup \{b\}X\{b\}$
3. $X = \{\epsilon\} \cup \{a\}\{a\} \cup XX$

corresponding to the three grammars mentioned above all have least solutions. These solutions are the languages defined by the grammars.

The Schroder-Bernstein Theorem. As a set-theoretic application of the Fixed Point Theorem we offer the proof of the following:

Theorem 2 (Schroder-Bernstein) *Let S and T be sets. If $f : S \to T$ and $g : T \to S$ are injections, then there is a bijection $h : S \to T$.*

Proof: The function $Y \mapsto (T - f^*(S)) \cup f^*(g^*(Y))$ from $\mathcal{P}T$ to $\mathcal{P}T$ is easily seen to be continuous with respect to the inclusion ordering. Hence, by the Fixed Point Theorem, there is a subset

$$Y = (T - f^*(S)) \cup f^*(g^*(Y)).$$

In particular, $T - Y = f^*(S - g^*(Y))$ since

$$\begin{aligned} T - Y &= T - ((T - f^*(S)) \cup f^*(g^*(Y))) \\ &= (T - (T - f^*(S))) \cap (T - (f^*(g^*(Y)))) \\ &= f^*(S) \cap (T - (f^*(g^*(Y)))) \\ &= f^*(S - g^*(Y)) \end{aligned}$$

Now define $h : S \to T$ by

$$h(x) = \begin{cases} y & \text{if } x = g(y) \text{ for some } y \in Y \\ f(x) & \text{otherwise} \end{cases}$$

This makes sense because g is an injection. Moreover, h itself is an injection since f and g are injections. To see that it is a surjection, suppose $y \in T$. If $y \in Y$, then $h(g(y)) = y$. If $y \notin Y$, then $y \in f^*(S - g^*(Y))$, so $y = f(x) = h(x)$ for some x. Thus h is a bijection. ∎

1.2.3 Uniformity.

The question naturally arises as to why we take the *least* fixed point in order to get the meaning. In most instances there will be other choices. There are several answers to this question. First of all, it seems intuitively reasonable to take the least defined function satisfying a given recursive equation.

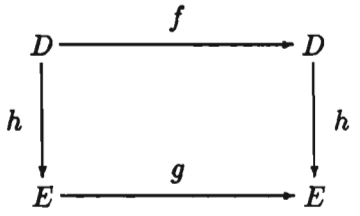
But more importantly, taking the least fixed point yields a *canonical* solution. Indeed, it is possible to show that, given a cpo D, the function $\mathsf{fix}_D : (D \to D) \to D$ given by $\mathsf{fix}_D(f) = \bigsqcup_n f^n(\bot)$ is actually *continuous.* But are there other operators like fix that could be used? A definition is helpful:

Definition: A *fixed point operator* F is a class of continuous functions

$$F_D : (D \to D) \to D$$

such that, for each cpo D and continuous function $f : D \to D$, we have $F_D(f) = f(F_D(f))$. ∎

Let us say that a fixed point operator F is *uniform* if, for any pair of continuous functions $f : D \to D$ and $g : E \to E$ and strict continuous function $h : D \multimap E$ which makes the following diagram commute



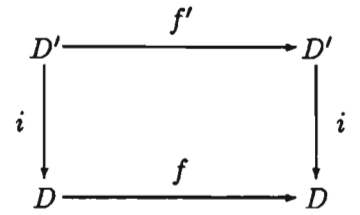
we have $h(F_D(f)) = F_E(g)$. We leave it to the reader to show that fix is a uniform fixed point operator. What is less obvious, and somewhat more surprising, is the following:

Theorem 3 fix *is the* unique *uniform fixed point operator.*

Proof: To see why this must be the case, let D be a cpo and suppose $f : D \to D$ is continuous. Then the set

$$D' = \{x \in D \mid x \sqsubseteq \mathsf{fix}(f)\}$$

is a cpo under the order that it inherits from the order on D. In particular, the restriction f' of f to D' has $\mathsf{fix}_D(f)$ as its *unique* fixed point. Now, if $i : D' \to D$ is the inclusion map then the following diagram commutes



Thus, if F is a uniform fixed point operator, we must have $F_D(f) = F_{D'}(f')$. But $F_{D'}(f')$ is a fixed point of f' and must therefore be equal to $\mathsf{fix}_D(f)$. ∎

We hope that these results go some distance toward convincing the reader that fix is a reasonable operator to use for the semantics of recursively defined functions.

1.3 Effectively presented domains.

There is a significant problem with the full class of cpo's as far as the theory of computation goes. There does not seem to be any reasonable way to define a general notion of *computable function* between cpo's. It is easy to see that these ideas make perfectly good sense for a noteworthy collection of examples. Consider a strict function $f : \mathsf{N}_\perp \multimap \mathsf{N}_\perp$. If we take $f(n) = \perp$ to mean that f is *undefined* at n, then f can be viewed as a partial function on N. We wish to have a concept of computability for functions on (some class of) cpo's so that f is computable just in case it corresponds to the usual notion of a partial recursive function. But we must also have a definition that applies to ***functionals,*** that is, functions which may take functions as arguments or return functions as values. We already encountered a functional earlier when we defined the factorial. To illustrate the point that there is a concept of computability that applies to such operators, consider, for example, a functional $F : (\mathsf{N}_\perp \multimap \mathsf{N}_\perp) \multimap \mathsf{N}_\perp$ which takes a function $f : \mathsf{N}_\perp \multimap \mathsf{N}_\perp$ and computes the value of f on the number 3. The functional F is continuous and it is *intuitively* computable. This intuition comes from the fact that, to compute $F(f)$ on an argument one needs only know how to compute f on an argument.

Our goal is to define a class of cpo's for which a notion of "finite approximation" makes sense. Let D be a cpo. An element $x \in D$ is *compact* if, whenever M is a directed subset of D and $x \sqsubseteq \bigsqcup M$, there is a point $y \in M$ such that $x \sqsubseteq y$. We let $K(D)$ denote the set of compact elements of D. The cpo D is said to be *algebraic* if, for every $x \in D$, the set $M = \{x_0 \in K(D) \mid x_0 \sqsubseteq x\}$ is directed and $\bigsqcup M = x$. In other words, in an algebraic cpo, each element is a directed limit of its "finite" (compact) approximations. If D is algebraic and $K(D)$ is countable, then we will say that D is a *domain.*

With the exception of the unit interval of real numbers, all of the cpo's we have mentioned so far are domains. The compact elements of the domain $\mathsf{N}_\perp \multimap \mathsf{N}_\perp$ are the functions with finite domain of definition, *i.e.* those continuous functions $f : \mathsf{N}_\perp \multimap \mathsf{N}_\perp$ such that $\{n \mid f(n) \neq \perp\}$ is finite. As another example, the collection $\mathcal{P}\mathsf{N}$ of subsets of N, ordered by subset inclusion is a domain whose compact elements are just the finite subsets of N.

One thing which makes domains particularly nice to work with is the way one may describe a continuous function $f : D \to E$ between domains D and E using the compact elements. Let G_f be the set of pairs (x_0, y_0) such that $x_0 \in K(D)$ and $y_0 \in K(E)$ and $y_0 \sqsubseteq f(x_0)$. If $x \in D$, then one may recover from G_f the value of f on x as

$$f(x) = \bigsqcup\{y_0 \mid (x_0, y_0) \in G_f \text{ and } x_0 \sqsubseteq x\}.$$

This allows us to characterize, for example, a continuous function $f : \mathcal{P}\mathsf{N} \to \mathcal{P}\mathsf{N}$ between *uncountable* cpo's with a ***countable*** set G_f. The significance of this fact for the theory of computability is not hard to see; we will say that the function f is computable just in case G_f is computable (in a sense to be made precise below).

1.3.1 Normal subposets and projections.

Before we give the formal definition of computability for domains and continuous functions, we digress briefly to introduce a useful relation on subposets. Given a poset $\langle A, \sqsubseteq \rangle$ and $x \in A$, let $\downarrow x = \{y \in A \mid y \sqsubseteq x\}$.

Definition: Let A be a poset and suppose $N \subseteq A$. Then N is said to be *normal* in A (and we write $N \lhd A$) if, for every $x \in A$, the set $N \cap \downarrow x$ is directed. ∎

The following lemma lists some useful properties of the relation $\lhd$.

Lemma 4 *Let C be a poset with a least element and suppose A and B are subsets of C.*

1. *If $A \lhd B \lhd C$ then $A \lhd C$.*
2. *If $A \subseteq B \subseteq C$ and $A \lhd C$ then $A \lhd B$.*
3. *If $A \lhd C$, then $\perp \in A$.*
4. *$\langle \mathcal{P}(C), \lhd \rangle$ is a cpo with $\{\perp\}$ as its least element.* ∎

Intuitively, a normal subposet $N \lhd A$ is an "approximation" to A. The notion of normal subposet is closely related to one of the central concepts in the theory of domains. A pair of continuous functions $g : D \to E$ and $f : E \to D$ is said to be an *embedding-projection* pair (g is the embedding and f is the projection) if they satisfy the following

$$f \circ g = \mathsf{id}_D$$
$$g \circ f \sqsubseteq \mathsf{id}_E$$

where id_D and id_E are the identity functions on D and E respectively (in future, we drop the subscripts when D and E are clear from context) and composition of functions is defined by $(f \circ g)(x) = f(g(x))$. One can show that each of f and g uniquely determines the other. Hence it makes sense to refer to f as the projection *determined by* g and refer to g as the embedding *determined by* f. There is quite a lot to be said about properties of projections and embeddings and we cannot begin to provide, in the space of this chapter, the full discussion that these concepts deserve (the reader may consult Chapter 0 of [GHK*80] for this). However, a few observations will be essential to what follows. We first provide a simple example:

Example: If $f : D \to E$ is a continuous function then there is a strict continuous function $\mathsf{strict} : (D \to E) \to (D \multimap E)$ given by:

$$\mathsf{strict}(f)(x) = \begin{cases} f(x) & \text{if } x \neq \perp \\ \perp & \text{if } x = \perp \end{cases}$$

The function **strict** is a projection whose corresponding embedding is the inclusion map $\mathsf{incl} : (D \multimap E) \hookrightarrow (D \to E)$. ∎

In our discussion below we will not try to make much of the distinction between $f : D \circ\!\!\rightarrow E$ and $\mathsf{incl}(f) : D \rightarrow E$ (for example, we may write $\mathsf{id} : D \circ\!\!\rightarrow D$ as well as $\mathsf{id} : D \rightarrow D$ or even $\mathsf{incl}(\mathsf{id}) : D \rightarrow D$). From the two equations that define the relationship between a projection and embedding, it is easy to see that a projection is a surjection (*i.e.* onto) and an embedding is an injection (*i.e.* one-to-one). Thus one may well think of the image of an embedding $g : D \rightarrow E$ as a special kind of sub-cpo of E. We shall be especially interested in the case where an embedding is an *inclusion* as in the case of $D \circ\!\!\rightarrow E$ and $D \rightarrow E$. Let D be a cpo. We say that a continuous function $p : D \rightarrow D$ is a *finitary projection* if $p \circ p = p \sqsubseteq \mathsf{id}$ and $\mathsf{im}(p) = \{p(x) \mid x \in D\}$ is a domain. Note, in particular, that the inclusion map from $\mathsf{im}(p)$ into D is an embedding (which has the corestriction of p to its image as the corresponding projection). It is possible to characterize the basis of $\mathsf{im}(p)$ as follows:

Lemma 5 *If D is a domain and $p : D \rightarrow D$ is a finitary projection, then the set of compact elements of $\mathsf{im}(p)$ is just $\mathsf{im}(p) \cap K(D)$. Moreover, $\mathsf{im}(p) \cap K(D) \lhd K(D)$.* ∎

Suppose, on the other hand, that $N \lhd K(D)$. Then it is easy to check that the function $p_N : D \rightarrow D$ given by

$$p_N(x) = \bigsqcup\{y \in N \mid y \sqsubseteq x\}$$

is a finitary projection. Indeed, the correspondence $N \mapsto p_N$ is inverse to the correspondence $p \mapsto \mathsf{im}(p) \cap K(D)$ and we have the following:

Theorem 6 *For any domain D there is an isomorphism between the cpo of normal substructures of $K(D)$ and the poset $\mathsf{Fp}(D)$ of finitary projections on D.* ∎

In particular, if $M \subseteq \mathsf{Fp}(D)$ is directed then $\mathsf{im}(\bigsqcup M)$ is a domain. This is a fact which will be significant later. Indeed, the notions of projection and normal subposet will come up again and again throughout the rest of our discussion.

1.3.2 Effectively presented domains.

Returning now to the topic of computability, we will say that a domain is effectively presented if the ordering on its basis is decidable and it is possible to effectively recognize the finite normal subposets of the basis:

Definition: Let D be a domain and suppose $d : \mathsf{N} \rightarrow K(D)$ is a surjection. Then d is an *effective presentation* of D if

1. the set $\{(m, n) \mid d_m \sqsubseteq d_n\}$ is effectively decidable, and
2. for any finite set $u \subseteq \mathsf{N}$, it is decidable whether $\{d_n \mid n \in u\} \lhd K(D)$.

If $\langle D, d\rangle$ and $\langle E, e\rangle$ are effectively presented domains, then a continuous function $f : D \rightarrow E$ is said to be *computable* (with respect to d and e) if and only if, for every $n \in \mathsf{N}$, the set $\{m \mid e_m \sqsubseteq f(d_n)\}$ is recursively enumerable. ∎

Unfortunately, the full class of domains has a serious problem. It is this: there are domains D, E such that the cpo $D \to E$ is *not* a domain (we will return to this topic in Section 1.6). Since we wish to use $D \to E$ in defining computability at higher types, we need some restriction on domains D and E which will insure that $D \to E$ is a domain. There are several restrictions which will work. We begin by presenting one which is relatively simple. Another will be discussed later.

Definition: A poset A is said to be *bounded complete* if A has a least element and every bounded subset of A has a least upper bound. ∎

The bounded complete domains are closely related to a more familiar class of cpo's which arise in many places in classical mathematics. A domain D is a (countably based) *algebraic lattice* if every subset of D has a least upper bound. It is not hard to see that a domain D is bounded complete if and only if the cpo $D^\top$ which results from adding a new top element to D is an algebraic lattice. The poset $\mathcal{P}\mathbf{N}$ is an example of an algebraic lattice. On the other hand, the bounded complete domain $\mathbf{N}_\perp \multimap \mathbf{N}_\perp$ lacks a top element and therefore fails to be an algebraic lattice. All of the domains we have discussed so far are bounded complete. In particular, we have the following:

Theorem 7 *If D and E are bounded complete domains, then $D \to E$ is also a bounded complete domain. Moreover, if D and E have effective presentations, then $D \to E$ has an effective presentation as well. Similar facts hold for $D \multimap E$.*

Proof: (Sketch) It is not hard to see that $D \to E$ is a bounded complete cpo whenever E is. To prove that $D \to E$ is a domain we must demonstrate its basis. Suppose $N \lhd K(D)$ is finite and $s : N \to K(E)$ is monotone. Then the function $\mathbf{step}(s) : D \to E$ given by taking $\mathbf{step}(s)(x) = \bigsqcup\{f(y) \mid y \in N \cap {\downarrow} x\}$ is continuous and compact in the ordering on $D \to E$. These are called *step functions* and it is possible to show that they form a basis for $D \to E$. The proof that the poset of step functions has decidable ordering and finite normal subposets is tedious, but not difficult, using the effective presentations of D and E. The proof of these facts for $D \multimap E$ is essentially the same since the strict step functions form a basis. ∎

In the remaining sections of the chapter we will discuss a great many operators like $\cdot \to \cdot$ and $\cdot \multimap \cdot$. We will leave it to the reader to convince himself that all of these operators preserve the property of having an effective presentation. Further discussion of computability on domains may be found in [Smy77] and [KT84]. It is hoped that future research in the theory of domains will provide a general technique which will incorporate computability into the *logic* whereby we reason about the existence of our operators. This will eliminate the need to provide demonstrations of effective presentations. This is a central idea in current investigations but it is beyond our scope to discuss it further.

1.4 Operators and functions.

There are a host of operators on domains which are needed for the purposes of semantic definitions. In this section we mention a few of them. An essential technique for building new operators from those which we present here will be introduced below when we discuss solutions of recursive equations.

1.4.1 Products.

Given posets D and E, the *product* $D \times E$ is the set of pairs (x, y), where $x \in D$ and $y \in E$. The ordering is coordinatewise, *i.e.* $(x, y) \sqsubseteq (x', y')$ if and only if $x \sqsubseteq x'$ and $y \sqsubseteq y'$. We define functions $\mathsf{fst} : D \times E \to D$ and $\mathsf{snd} : D \times E \to E$ given by $\mathsf{fst}(x, y) = x$ and $\mathsf{snd}(x, y) = y$. If a subset $L \subseteq D \times E$ is directed, then

$$M = \mathsf{fst}^*(L) = \{x \mid \exists y \in E.\ (x, y) \in L\}$$
$$N = \mathsf{snd}^*(L) = \{y \mid \exists x \in D.\ (x, y) \in L\}$$

are directed. In particular, if D and E are cpo's, then $\bigsqcup L = (\bigsqcup M, \bigsqcup N)$ and, of course, $\bot_{D \times E} = (\bot_D, \bot_E)$, so $D \times E$ is a cpo. Indeed, if D and E are domains, then $D \times E$ is also a domain with $K(D \times E) = K(D) \times K(E)$. The property of bounded completeness is also preserved by $\times$.

Given cpos D, E, F, one can show that a function $f : D \times E \to F$ is continuous if and only if it is continuous in each of its arguments individually. In other words, f is continuous iff each of the following conditions holds:

1. For every directed set $M \subseteq D$ and element $e \in E$, the function $f_1 : D \to F$ given by $x \mapsto f(x, e)$ is continuous.
2. For every directed set $N \subseteq E$ and element $d \in D$, the function $f_2 : E \to F$ given by $y \mapsto f(d, y)$ is continuous.

We leave the proof of this equivalence as an exercise for the reader.

It is easy to see that each of the functions **fst** and **snd** is continuous. Moreover, given any cpo F and continuous functions $f : F \to D$ and $g : F \to E$, there is a continuous function $\langle f, g \rangle : F \to D \times E$ such that

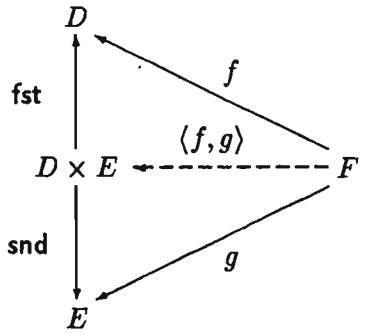
$$\mathsf{fst} \circ \langle f, g \rangle = f$$
$$\mathsf{snd} \circ \langle f, g \rangle = g$$

and, for any continuous function $h : F \to D \times E$,

$$\langle \mathsf{fst} \circ h, \mathsf{snd} \circ h \rangle = h.$$

The function $\langle f, g \rangle$ is given by $\langle f, g \rangle(x) = (f(x), g(x))$.

There is another, more pictorial, way of stating these equational properties of the operator $\langle \cdot, \cdot \rangle$ using a commutative diagram. The desired property can be stated in the following manner: given any cpo F and continuous functions $f : F \to D$ and $g : F \to E$, there is a *unique* continuous function $\langle f, g \rangle$ which completes the following diagram:



This is referred to as the *universal property* of the operator $\times$. As operators are given below we will describe the universal properties that they satisfy and these will form the basis of a system of equational reasoning about continuous functions. Virtually all of the functions needed to describe the semantics of (a wide variety of) programming languages may be built from those which are used in expressing these universal properties!

Given continuous functions $f : D \to D'$ and $g : E \to E'$, we may define a continuous function $f \times g$ which takes (x, y) to $(f(x), g(y))$ by setting

$$f \times g = \langle f \circ \mathsf{fst}, g \circ \mathsf{snd} \rangle : D \times E \to D' \times E'.$$

It is easy to show that $\mathsf{id}_D \times \mathsf{id}_E = \mathsf{id}_{D \times E}$ and

$$(f \times g) \circ (f' \times g') = (f \circ f') \times (g \circ g').$$

Note that we have "overloaded" the symbol $\times$ so that it works both on pairs of *domains* and pairs of *functions*. This sort of overloading is quite common in mathematics and we will use it often below. In this case (and others to follow) we have an example of what mathematicians call a *functor*.

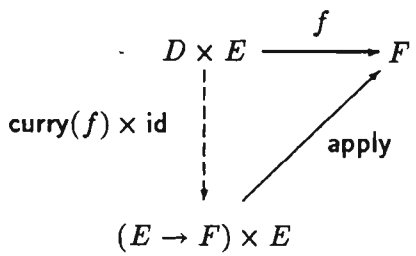
There is a very important relationship between the operators $\to$ and $\times$. Let D, E and F be cpo's. Then there is a function

$$\mathsf{apply} : ((E \to F) \times E) \to F$$

given by taking $\mathsf{apply}(f, x)$ to be $f(x)$ for any function $f : E \to F$ and element $x \in E$. Indeed, the function apply is *continuous*. Also, given a function $f : D \times E \to F$, there is a continuous function

$$\mathsf{curry}(f) : D \to (E \to F)$$

given by taking $\mathsf{curry}(f)(x)(y)$ to be $f(x, y)$. Moreover, $\mathsf{curry}(f)$ is the *unique* continuous function which makes the following diagram commute:



This uniqueness condition is equivalent to the following equation:

$$\mathsf{curry}(\mathsf{apply} \circ (h \times \mathsf{id}_E)) = h \tag{1.1}$$

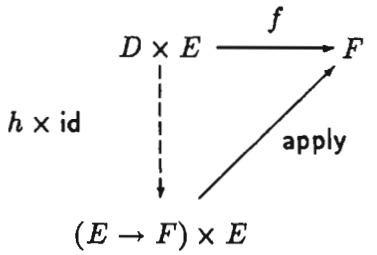
To see this, suppose equation (1.1) holds and h satisfies

$$f = \mathsf{apply} \circ (h \times \mathsf{id})$$

then

$$\mathsf{curry}(f) = \mathsf{curry}(\mathsf{apply} \circ (h \times \mathsf{id})) = h$$

so the uniqueness condition is satisfied. On the other hand, if $\mathsf{curry}(f)$ is uniquely determined by the diagram above, then equation (1.1) follows immediately from the commutativity of the following diagram:



for $f = \mathsf{apply} \circ (h \times \mathsf{id}_E)$.

It is often useful to have a multiary notation for products. We write

$$\begin{aligned} \times() &= \mathsf{I} \\ \times(D_1, \ldots, D_n) &= \times(D_1, \ldots, D_{n-1}) \times D_n \end{aligned}$$

and define projections

$$\mathsf{on}_i : \times(D_1, \ldots, D_n) \to D_i$$

by

$$\mathsf{on}_i = \mathsf{snd} \circ \mathsf{fst}^{n-i}$$

Similarly, one defines a multiary version of the pairing operation by taking $\langle\,\rangle$ to be the identity on the one point domain and defining

$$\langle f_1, \ldots, f_n \rangle = \langle \langle f_1, \ldots, f_{n-1} \rangle, f_n \rangle.$$

These multiary versions of projection and pairing satisfy a universal property similar to the one for the binary product.

1.4.2 Church's λ-notation.

If we wish to define a function from, say, natural numbers to natural numbers, we typically do so by describing the action of that function on a generic number x (a *variable*) using previously defined functions. For example, the squaring function f has the action $x \mapsto x * x$ where $*$ is the multiplication function. We may now use f to define other functions: for example, a function g which takes a function $h : \mathsf{N} \to \mathsf{N}$ to $f \circ h$. Continuing in this way we may construct increasingly complex function definitions. However, it is sometimes useful to have a notation for functions which alleviates the necessity of introducing intermediate names. This purpose is served by a terminology known as λ-notation which is originally due to Church.

The idea is this. Instead of introducing a term such as f and describing its action as a function, one simply gives the function a name which is basically a description of what it does with its argument. In the above case one writes $\lambda x.\ x * x$ for f and $\lambda h.\ f \circ h$ for g. One can use this notation to define g without introducing f by defining g to be the function $\lambda h.\ (\lambda x.\ x * x) \circ h$. The λh at the beginning of this expression says that g is a function which is computed by taking its argument and *substituting* it for the variable h in the expression $(\lambda x.\ x * x) \circ h$.

The use of the Greek letter λ for the operator which binds variables is primarily an historical accident. Various programming languages incorporate something essentially equivalent to λ-notation using other names. In mathematics textbooks it is common to avoid the use of such notation by assuming conventions about variable names. For example, one may write

$$x^2 - 2 * x$$

for the function which takes a real number as an argument and produces as result the square of that number less its double. An expression such as

$$x^2 + x * y + y^2$$

would denote a function which takes two numbers as arguments—that is, the values of x and y—and produces the square of the one number plus the square of the other plus the product of the two. One might therefore provide a name for this function by writing something like:

$$f(x, y) = x^2 + x * y + y^2.$$

So f is a function which takes a pair of numbers and produces a number. But what notation should we use for the function g that takes a number n as argument and produces the *function* $n \mapsto x^2 + x * n + n^2$? For example, $g(2)$ is the function $x^2 + 2 * x + 4$. It is not hard to see that this is closely related to the function curry which we discussed above. Modulo the fact that we defined curry for domains above, we might have written $g = \mathsf{curry}(f)$. Or, to define g directly, we would write

$$g = \lambda y.\ \lambda x.\ x^2 + x * y + y^2.$$

The definition of f would need to be given differently since f takes a *pair* as an argument. We therefore write:

$$f = \lambda(x, y).\ x^2 + x * y + y^2.$$

There is no impediment to using this notation to describe higher-order functions as well. For example,

$$\lambda f.\ f(3)$$

takes a function f and evaluates it on the number 3 and

$$\lambda f.\ f \circ f$$

takes a function and composes it with itself. But these definitions highlight a very critical issue. Note that both definitions are *ambiguous* as they stand. Does the function $\lambda f.\ f(3)$ take, for example, functions from numbers to reals as argument or does it take a function from numbers to sets of numbers as argument? Either of these would, by itself, make sense. What we need to do is indicate somewhere in the expression the *types* of the variables (and constants if their types are not already understood). So we might write

$$\lambda f : \mathsf{N} \to \mathsf{R}.\ f(3)$$

for the operator taking a real valued function as argument and

$$\lambda f : \mathsf{N} \to \mathcal{P}\mathsf{N}.\ f(3)$$

for the operator taking a $\mathcal{P}\mathsf{N}$ valued function.

So far, what we have said applies to almost any class of spaces and functions where products and an operator like curry are defined. But for the purposes of programming semantics, we need a semantic theory that includes the concept of a fixed point. Such fixed points are guaranteed if we stay within the realm of cpo's and continuous functions. But the crucial fact is this: *the process of λ-abstraction preserves continuity.* This is because $\mathsf{curry}(f)$ is continuous whenever f is. We may therefore use the notational tools we have described above with complete freedom and still be sure that recursive definitions using this notation make sense.

Demonstrating that the typed λ-calculus (*i.e.* the system of notations that we have been describing informally here) is really useful in explaining the semantics of programming languages is not the objective of this chapter. However, one can already see that it provides a considerable latitude for writing function definitions in a simple and mathematically perspicuous manner.

1.4.3 Smash products.

In the product $D \times E$ of cpo's D and E, there are elements of the form $(x, \perp)$ and $(\perp, y)$. If $x \neq \perp$ or $y \neq \perp$, then these will be *distinct* members of $D \times E$. In programming semantics, there are occasions when it is desirable to *identify* the pairs $(x, \perp)$ and $(\perp, y)$. For this purpose, there is a collapsed version of the product called the *smash product.* For cpo's D and E, the smash product $D \otimes E$ is the set

$$\{(x, y) \in D \times E \mid x \neq \perp \text{ and } y \neq \perp\} \cup \{\perp_{D \otimes E}\}$$

where $\perp_{D\otimes E}$ is some new element which is not a pair. The ordering on pairs is coordinatewise and we stipulate that $\perp_{D\otimes E} \sqsubseteq z$ for every $z \in D \otimes E$. There is a continuous surjection

$$\textsf{smash} : D \times E \to D \otimes E$$

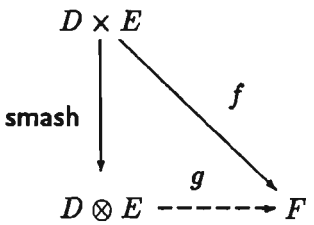
given by taking

$$\textsf{smash}(x,y) = \begin{cases} (x,y) & x \neq \perp \text{ and } x \neq \perp \\ \perp_{D\otimes E} & \text{otherwise} \end{cases}$$

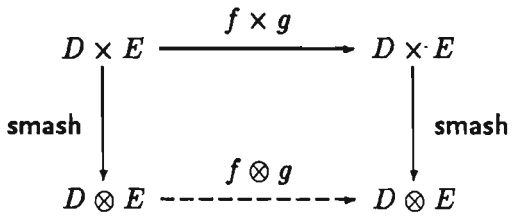
This function establishes a useful relationship between $D \times E$ and $D \otimes E$. In fact, it is a projection whose corresponding embedding is the function $\textsf{unsmash} : D \otimes E \to D \times E$ given by

$$\textsf{unsmash}(z) = \begin{cases} z & \text{if } z = (x,y) \text{ is a pair} \\ (\perp,\perp) & \text{if } z = \perp_{D\otimes E} \end{cases}$$

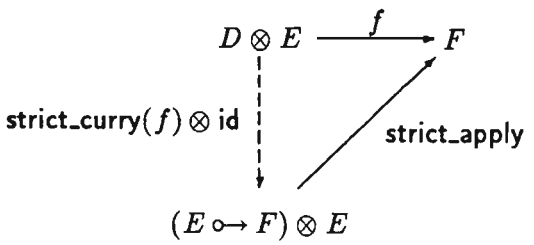
Let us say that a function $f : D \times E \to F$ is *bistrict* if $f(x,y) = \perp$ whenever $x = \perp$ or $y = \perp$. If $f : D \times E \to F$ is bistrict and continuous, then $g = f \circ \textsf{unsmash}$ is the unique strict, continuous function which completes the following diagram:



If $f : D \to D'$ and $g : E \to E'$ are strict continuous functions, then $f \otimes g = \textsf{smash} \circ (f \times g) \circ \textsf{unsmash}$ is the unique strict, continuous function which completes the following diagram:



As with the product $\times$ and function space $\to$, there is a relationship between the smash product $\otimes$ and the strict function space $\multimap$. In particular, there is a strict continuous function strict_apply such that for any strict function f, there is a unique strict function strict_curry such that the following diagram commutes:



1.4.4 Sums and lifts.

Given cpo's D and E, we define the *coalesced sum* $D \oplus E$ to be the set

$$\Big((D - \{\bot_D\}) \times \{0\}\Big) \cup \Big((E - \{\bot_E\}) \times \{1\}\Big) \cup \{\bot_{D\oplus E}\}$$

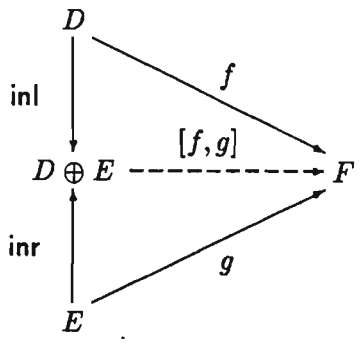
where $D - \{\bot_D\}$ and $E - \{\bot_E\}$ are the sets D and E with their respective bottom elements removed and $\bot_{D\oplus E}$ is a new element which is not a pair. It is ordered by taking $\bot_{D\oplus E} \sqsubseteq z$ for all $z \in D \oplus E$ and taking $(x, m) \sqsubseteq (y, n)$ if and only if $m = n$ and $x \sqsubseteq y$. There are strict continuous functions $\mathsf{inl} : D \multimap (D \oplus E)$ and $\mathsf{inr} : E \multimap (D \oplus E)$ given by taking

$$\mathsf{inl}(x) = \begin{cases} (x, 0) & \text{if } x \neq \bot \\ \bot_{D\oplus E} & \text{if } x = \bot \end{cases}$$

and

$$\mathsf{inr}(x) = \begin{cases} (x, 1) & \text{if } x \neq \bot \\ \bot_{D\oplus E} & \text{if } x = \bot \end{cases}$$

Moreover, if $f : D \multimap F$ and $g : E \multimap F$ are strict continuous functions, then there is a unique strict continuous function $[f, g]$ which completes the following diagram:



The function $[f, g]$ is given by

$$[f, g](z) = \begin{cases} f(x) & \text{if } z = (x, 0) \\ g(y) & \text{if } z = (y, 1) \\ \bot & \text{if } z = \bot. \end{cases}$$

Given continuous functions $f : D \multimap D'$ and $g : E \multimap E'$, we define

$$f \oplus g = [\mathsf{inl} \circ f, \mathsf{inr} \circ g] : D \oplus E \multimap D' \oplus E'.$$

As with the product, it is useful to have a multiary notation for the coalesced sum. We define

$$\oplus() = \mathsf{I}$$
$$\oplus(D_1, \ldots, D_n) = \oplus(D_1, \ldots, D_{n-1}) \oplus D_n$$

and

$$\mathsf{in}_i = \mathsf{inr} \circ \mathsf{inl}^{n-i}.$$

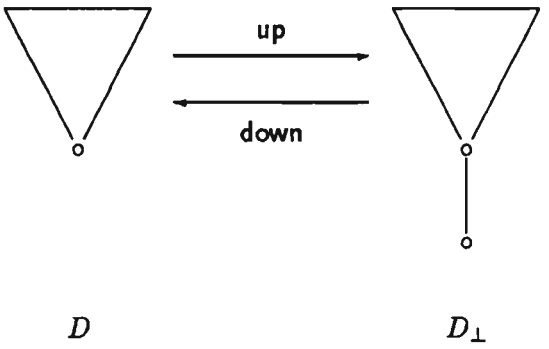


Figure 1.2: The lift of a cpo.

One may also define $[f_1, \ldots, f_n]$ and prove a universal property.

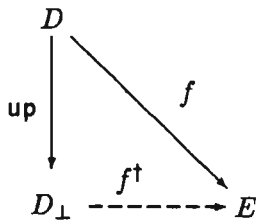
Given a cpo D, we define the *lift* of D to be the set $D_\perp = (D \times \{0\}) \cup \{\perp\}$, where $\perp$ is a new element which is not a pair, together with an partial ordering $\sqsubseteq$ which is given by stipulating that $(x,0) \sqsubseteq (y,0)$ when $x \sqsubseteq y$ and $\perp \sqsubseteq z$ for every $z \in D_\perp$. In short, $D_\perp$ is the poset obtained by adding a new bottom to D—see Figure 1.2. It is easy to show that $D_\perp$ is a cpo if D is. We define a strict continuous function $\mathsf{down} : D_\perp \multimap D$ by

$$\mathsf{down}(z) = \begin{cases} x & \text{if } z = (x,0) \\ \perp_D & \text{otherwise} \end{cases}$$

and a (non-strict) continuous function $\mathsf{up} : D \to D_\perp$ given by $\mathsf{up} : x \mapsto (x,0)$. These functions are related by

$$\mathsf{down} \circ \mathsf{up} = \mathsf{id}_D$$
$$\mathsf{up} \circ \mathsf{down} \sqsupseteq \mathsf{id}_{D_\perp}$$

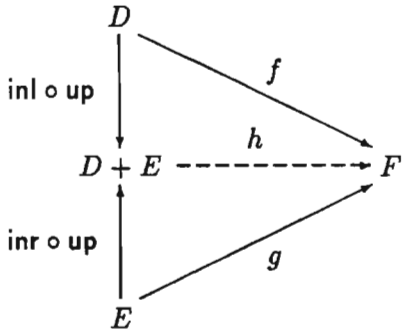
These inequations are reminiscent of those which we gave for embedding-projection pairs, but the second inequation has $\sqsupseteq$ rather than $\sqsubseteq$. We will discuss such pairs of functions later. Given cpo's D and E and continuous function $f : D \to E$, there is a unique strict continuous function $f^\dagger$ which completes the following diagram:



Given a continuous function $f : D \to E$, we define a strict continuous function

$$f_\perp = (\mathsf{up} \circ f)^\dagger : D_\perp \multimap E_\perp.$$

Given cpo's D and E, we define the *separated sum* $D + E$ to be the cpo $D_\perp \oplus E_\perp$. By the universal properties for $\oplus$ and $(\cdot)_\perp$, we know that $h = [f^\dagger, g^\dagger]$ is the unique *strict* continuous function which completes the following diagram:



However, h may not be the only *continuous* function which completes the diagram. Given continuous functions $f : D \to D'$ and $g : E \to E'$, we define

$$f + g = f_\perp \oplus g_\perp : D + E \to D' + E'.$$

1.4.5 Isomorphisms and closure properties.

There are quite a few interesting relationships between the operators above which are implied by the definitions and commutative diagrams. We list a few of these in the following lemmas.

Lemma 8 *Let D, E and F be cpo's, then*

1. $D \times E \cong E \times D$,
2. $(D \times E) \times F \cong D \times (E \times F)$,
3. $D \to (E \times F) \cong (D \to E) \times (D \to F)$,
4. $D \to (E \to F) \cong (D \times E) \to F$. ∎

Lemma 9 *Let D, E and F be cpo's, then*

1. $D \otimes E \cong E \otimes D$,
2. $(D \otimes E) \otimes F \cong D \otimes (E \otimes F)$,
3. $(E \oplus F) \multimap D \cong (E \multimap D) \times (E \multimap F)$,
4. $D \multimap (E \multimap F) \cong (D \otimes E) \multimap F$,
5. $D \otimes (E \oplus F) \cong (D \otimes E) \oplus (D \otimes E)$
6. $D_\perp \multimap E \cong D \to E$. ∎

We remarked already that $D \to E$ and $D \multimap E$ are bounded complete domains whenever D and E are. It is not difficult to see that similar closure properties will hold for the other operators we have defined in this section:

Lemma 10 *If D and E are bounded complete domains then so are the cpo's $D \to E$, $D \multimap E$, $D \times E$, $D \otimes E$, $D + E$, $D \oplus E$, $D_\perp$.* ∎

Further discussion of the operators defined in this section and others may be found in [Sco82a] and [Sco82b].

1.5 Powerdomains.

We now turn our attention to another collection of operators on domains. Just as we have defined a computable analog to the *function space,* we will now define a computable analog to the *powerset operation.* Actually, we will produce three such operators. In the domain theory literature these are called *powerdomains.* If D is a domain we write

- $D^\sharp$ for the *upper* powerdomain of D,
- $D^\natural$ for the *convex* powerdomain of D, and
- $D^\flat$ for the *lower* powerdomain of D.

The names we use for these operators come from the concepts of upper and lower semi-continuity and the interested reader can consult [Smy83b] for a detailed explanation. They commonly appear under other names as well. The convex powerdomain $D^\natural$ was introduced by Gordon Plotkin [Plo76] and is therefore sometimes referred to as the *Plotkin powerdomain.* The upper powerdomain $D^\sharp$ was introduced by Mike Smyth [Smy78] and is sometimes called the *Smyth powerdomain.* For reasons that we will discuss briefly below, this latter powerdomain corresponds to the *total correctness* interpretation of programs. Since Tony Hoare has done much to popularize the study of *partial correctness* properties of programs, the remaining powerdomain $D^\flat$—which corresponds to the partial correctness interpretation—sometimes bears his name.

1.5.1 Intuition.

There is a basic intuition underlying the powerdomain concept which can be explained through the concept of *partial information.* To keep things simple, let us assume that we are given a finite poset A and asked to form the *poset* of finite non-empty subsets of A. As a first guess, one might take the non-empty subsets and order them by subset inclusion. However, this operation ignores the order structure on A! Think of A as a collection of partial descriptions of data elements: $x \sqsubseteq y$ just in case x is a partial description of y. What should it mean for one non-empty subset of A to be a "partial description" of another? The are at least three reasonable philosophies that one might adopt in attempting to answer this question.

Suppose, for example, that I hold a bag of fruit and I wish to give you information about what is in the bag. One such description might be

A fruit in the bag is a yellow fruit or a red fruit.

This description is based on two basic pieces of data: "is a yellow fruit" and "is a red fruit". These are used to *restrict* the kinds of fruit which are in the bag. A more informative description of this kind would provide further restrictions. Consider the following example:

A fruit in the bag is a yellow fruit or a cherry or a strawberry.

It is based on three pieces of data: "is a yellow fruit", "is a cherry" and "is a strawberry". Since these three data provide further restrictions on the contents of the bag (by ruling out the possibility of an apple, for example) it is a more informative statement about the bag's contents. On the other hand,

A fruit in the bag is a yellow fruit or a red fruit or a purple fruit.

is a less informative description because it is more permissive; for instance, it does not rule out the possibility that the bag holds a grape. Now suppose that u, v are subsets of the poset A from the previous paragraph. With this way of seeing things, we should say that u is below v if the restrictions imposed by v are refinements of the restrictions imposed by u: that is, for each $y \in v$, there is an $x \in u$ such that $x \sqsubseteq y$. This is the basic idea behind the *upper powerdomain* of A.

Returning to the bag of fruit analogy, we might view the following as a piece of information about the contents of the bag

There is some yellow fruit and some red fruit in the bag.

This information is based on two pieces of data: "is a yellow fruit" and "is a red fruit". However, these data are not being used as before. They do not restrict possibilities; instead they offer a *positive* assertion about the contents of the bag. A more informative description of this kind would provide a further enumeration and refinement of the contents:

There is a banana, a cherry and some purple fruit in the bag.

This refined description does not rule out the possibility that the bag holds a apple, but it does insure that there is an cherry. A statment such as

There is some yellow fruit in the bag.

is less informative since it does not mention the presence of red fruit. Now suppose that u, v are subsets of the poset A. With this way of seeing things, we should say that u is below v if the positive assertions provided by u are extended and refined by v: that is, for each $x \in u$, there is a $y \in v$ such that $x \sqsubseteq y$. This is the basic idea behind the *lower powerdomain* of A.

Now, the *convex powerdomain* combines these two forms of information. For example, the assertion

If you pull a fruit from the bag, then it must be yellow or a cherry, and you can pull a yellow fruit from the bag and you can pull a cherry from the bag.

is this combined kind of information. The pair of assertions means that the bag holds some yellow fruit and at least one cherry, but nothing else. A more refined description might be

If you pull a fruit from the bag, then it must be a banana or a cherry, and you can pull a banana from the bag and you can pull a cherry from the bag.

A less refined description might be

If you pull a fruit from the bag, then it must be yellow or red, and you can pull a yellow fruit from the bag and you can pull a red fruit from the bag.

The reader may be curious about what bags of fruit have to do with programming semantics. The powerdomains are used to model non-deterministic computations where one wishes to speak about the set of outcomes of a computation. How one wishes to describe such outcomes will determine which of the three powerdomains is used. We will attempt to illustrate this idea later in this section—when we have given some formal definitions.

1.5.2 Formal definitions.

In order to give the definitions of the powerdomains, it is helpful to have a little information about the representation of domains using the concept of a pre-order:

Definition: A *pre-order* is a set A together with a binary relation $\vdash$ which is reflexive and transitive.

It is conventional to think of the relation $a \vdash b$ as indicating that a is "larger" than b (as in mathematical logic, where $\phi \vdash \psi$ means that the formula ψ follows from the hypothesis ϕ). Of course, any poset is also a pre-order. On the other hand, a pre-order may fail to be a poset by not satisfying the anti-symmetry axiom. In other words, we may have $x \vdash y$ and $y \vdash x$ but $x \neq y$. By identifying elements x, y which satisfy $x \vdash y$ and $y \vdash x$, we obtain an induced partially ordered set from a pre-order (and this why they are called *pre*-orders). We shall be particularly interested in a special kind of subset of a pre-order:

Definition: An *ideal* over a pre-order $\langle A, \vdash \rangle$ is a subset $s \subseteq A$ such that

1. if $u \subseteq s$ is finite, then there is an $x \in s$ such that $x \vdash y$ for each $y \in u$, and
2. if $x \in s$ and $x \vdash y$, then $y \in s$. ∎

In short, an ideal is a subset which is directed and downward closed. If $x \in A$ for a pre-order A, then the set

$$\downarrow x = \{y \in A \mid x \vdash y\}$$

is an ideal called the *principal ideal generated by* x. To induce a poset from a pre-order, one can take the poset of principal ideals under set inclusion. The poset of all ideals on a pre-order is somewhat more interesting:

Theorem 11 *Given a countable pre-order $\langle A, \vdash \rangle$, let D be the poset consisting of the ideals over A, ordered by set inclusion. If there is an element $\perp \in A$ such that $x \vdash \perp$ for each $x \in A$, then D is a domain and $K(D)$ is the set of principal ideals over A.*

Proof: Clearly, the ideals of A form a poset under set inclusion and the principal ideal $\downarrow \perp$ is the least element. To see that this poset is complete, suppose that $M \subseteq D$ and let $x = \bigcup M$. If we can show that x is an ideal, then it is certainly the least upper bound of M in D. To this end, suppose

$u \subseteq x$ is finite. Since each element of u must be contained in some element of M, there is a finite collection of ideals $s \subseteq M$ such that $u \subseteq \bigcup s$. Since M is directed, there is an element $y \in M$ such that $z \subseteq y$ for each $z \in s$. Thus $u \subseteq y$ and since y is ideal, there is an element $a \in y$ such that $b \sqsubseteq a$ for each $b \in u$. But $a \in y \subseteq x$, so it follows that x is an ideal.

To see that D is a domain, we show that the set of principal ideals is a basis. Suppose $M \subseteq D$ is directed and $\downarrow a \subseteq \bigcup M$ for some $a \in A$. Then $a \in x$ for some $x \in M$, so $\downarrow a \subseteq x$. Hence $\downarrow a$ is compact in D. Now suppose $x \in D$ and $u \subseteq A$ is a finite collection of elements of A such that $\downarrow a \subseteq x$ for each $a \in u$. Then $u \subseteq x$ and since x is an ideal, there is an element $b \in x$ with $b \vdash a$ for each $a \in u$. Thus $\downarrow a \subseteq \downarrow b$ for each $a \in u$ and it follows that the principal ideals below x form a directed collection. It is obvious that the least upper bound (*i.e.* union) of that collection is x. Since x was arbitrary, it follows that D is an algebraic cpo with principal ideals of A as its basis. Since A is countable, there are only countably many principal ideals, so D is a domain. ∎

For any set S, we let $\mathcal{P}_f^*(S)$ be the set of finite non-empty subsets of S. We write $\mathcal{P}_f(S)$ for the set of all finite subsets (including the empty set). Given a poset $\langle A, \sqsubseteq \rangle$, define a pre-ordering $\vdash^\sharp$ on $\mathcal{P}_f^*(A)$ as follows,

$$u \vdash^\sharp v \text{ if and only if } (\forall x \in u)(\exists y \in v).\ x \sqsupseteq y.$$

Dually, define a pre-ordering $\vdash^\flat$ on $\mathcal{P}_f^*(A)$ by

$$u \vdash^\flat v \text{ if and only if } (\forall y \in v)(\exists x \in u).\ x \sqsupseteq y.$$

And define $\vdash^\natural$ on $\mathcal{P}_f^*(A)$ by

$$u \vdash^\natural v \text{ if and only if } u \vdash^\sharp v \text{ and } u \vdash^\flat v.$$

If D is a domain, then let $D^\natural$ be the domain of ideals over $\langle \mathcal{P}_f^*(K(D)), \vdash^\natural \rangle$. We call $D^\natural$ the *convex powerdomain* of D. Similarly, define $D^\sharp$ and $D^\flat$ to be the domains of ideals over $\langle \mathcal{P}_f^*(K(D)), \vdash^\sharp \rangle$ and $\langle \mathcal{P}_f^*(K(D)), \vdash^\flat \rangle$ respectively. We call $D^\sharp$ the *upper powerdomain* of D and $D^\flat$ the *lower powerdomain* of D.

As an example, we compute the lower powerdomain of $\mathsf{N}_\perp$. Since $K(\mathsf{N}_\perp) = \mathsf{N}_\perp$, the lower powerdomain of $\mathsf{N}_\perp$ is the set of ideals over the pre-order $\langle \mathcal{P}_f^*(\mathsf{N}_\perp), \vdash^\flat \rangle$. To see what such an ideal must look like, note first that $u \vdash^\flat u \cup \{\perp\}$ and $u \cup \{\perp\} \vdash^\flat u$ for any $u \in \mathcal{P}_f^*(\mathsf{N}_\perp)$. From this fact it is already possible to see why $\vdash^\flat$ is usually only a *pre-order* and not a poset. Now, if u and v both contain $\perp$, then $u \vdash^\flat v$ iff $u \supseteq v$. Hence we may identify an ideal $x \in (\mathsf{N}_\perp)^\flat$ with the union $\bigcup x$ of all the elements in x. Thus $(\mathsf{N}_\perp)^\flat$ is isomorphic to the domain $\mathcal{P}\mathsf{N}$ of all subsets of N under subset inclusion.

Now let us compute the upper powerdomain of $\mathsf{N}_\perp$. Note that if u and v are finite non-empty subsets of $\mathsf{N}_\perp$ and $\perp \in v$, then $u \vdash^\sharp v$. In particular, any ideal x in $(\mathsf{N}_\perp)^\sharp$ contains all of the finite subsets v of $\mathsf{N}_\perp$ with $\perp \in v$. So, let us say that a set $u \in \mathcal{P}_f^*(\mathsf{N}_\perp)$ is *non-trivial* if it does not contain $\perp$ and an ideal $x \in (\mathsf{N}_\perp)^\sharp$ is non-trivial if there is a non-trivial $u \in x$. Now, if u and v are non-trivial, then $u \vdash^\sharp v$ iff $u \subseteq v$. Therefore, if an ideal x is non-trivial, then it is the principal ideal

generated by the intersection of its non-trivial elements! The smaller this set is, the larger is the ideal x. Hence, the non-trivial ideals in the powerdomain (ordered by subset inclusion) correspond to finite subsets of N (ordered by superset inclusion). If we now throw in the unique trivial ideal, we can see that $(\mathsf{N}_\perp)^\sharp$ is isomorphic to the domain of sets $\{\mathsf{N}\} \cup \mathcal{P}_f^*(\mathsf{N})$ ordered by superset inclusion.

Finally, let us look at the convex powerdomain of $\mathsf{N}_\perp$. If $u, v \in \mathcal{P}_f^*(\mathsf{N}_\perp)$, then $u \vdash^\natural v$ iff

1. $\perp \in v$ and $u \supseteq v$ or
2. $u = v$

Hence, if x is an ideal and there is a set $u \in x$ with $\perp \notin u$, then x is the principal ideal generated by u. No two distinct principal ideals like this will be comparable. On the other hand, if x is an ideal with $\perp \in u$ for each $u \in x$, then $x \subseteq y$ for an arbitrary ideal y iff $\bigcup x \subseteq \bigcup y$. Thus the convex powerdomain of $\mathsf{N}_\perp$ corresponds to the set of finite, non-empty subsets of N unioned with the set of arbitrary subsets of $\mathsf{N}_\perp$ that contain $\perp$. The ordering on these sets is like the pre-ordering $\vdash^\natural$ but extended to include infinite sets.

1.5.3 Universal and closure properties.

If $s, t \in D^\natural$ then we define a binary operation

$$s \uplus t = \{w \mid u \cup v \vdash^\natural w \text{ for some } u \in s \text{ and } v \in t\}.$$

This set is an ideal and the function $\uplus : D^\natural \times D^\natural \to D^\natural$ is continuous. Similar facts apply when $\uplus$ is defined in this way for $D^\sharp$ and $D^\flat$. Now, if $x \in D$, define

$$\{\!|x|\!\} = \{u \in \mathcal{P}_f^*(K(D)) \mid \{x_0\} \vdash^\natural u \text{ for some compact } x_0 \sqsubseteq x\}.$$

This forms an ideal and $\{\!|\cdot|\!\} : D \to D^\natural$ is a continuous function. When one replaces $\vdash^\natural$ in this definition by $\vdash^\sharp$ or $\vdash^\flat$, then similar facts apply. Strictly speaking, we should decorate the symbols $\uplus$ and $\{\!|\cdot|\!\}$ with indices to indicate their types, but this clutters the notation somewhat. Context will determine what is intended.

These three operators $(\cdot)^\sharp$, $(\cdot)^\flat$ and $(\cdot)^\natural$ may not seem to be the most obvious choices for the computable analog of the powerset operator. We will attempt to provide some motivation for choosing them in the remainder of this section. Given the operators $\uplus$ and $\{\!|\cdot|\!\}$, we may say that a point $x \in D$ for a domain D is an "element" of a set s in a powerdomain of D if $\{\!|x|\!\} \uplus s = s$. If s and t lie in a powerdomain of D, then s is a "subset" of t if $s \uplus t = t$. Care must be taken, however, not to confuse "sets" in a powerdomain with sets in the usual sense. The relations of "element" and "subset" described above will have different properties in the three different powerdomains. Moreover, it may be the case that s is a "subset" of t without it being the case that $s \subseteq t$!

To get some idea how the powerdomains are related to the semantics of non-deterministic programs, let us discuss non-deterministic partial functions from N to N. As we have noted before, there is a correspondence between partial functions from N to N and strict functions $f : \mathsf{N}_\perp \multimap \mathsf{N}_\perp$. These may be thought of as the meanings of "deterministic" programs, because the output of

a program is uniquely determined by its input (*i.e.* the meaning is a partial *function*). Suppose, however, that we are dealing with programs which permit some *finite non-determinism* as discussed in the section on non-determinism in the chapter of Peter Mosses. Then we may wish to think of a program as having as its meaning a function $f : \mathsf{N}_\perp \to P(\mathsf{N}_\perp)$ where P is one of the powerdomains. For example, if a program may give a 1 or a 2 as an output when given a 0 as input, then we will want the meaning f of this program to satisfy $f(0) = \{\!|1|\!\} \uplus \{\!|2|\!\} = \{\!|1,2|\!\}$. The three different powerdomains reflect three different views of how to relate the various possible program behaviors in the case of divergence. The upper powerdomain identifies program behaviors which *may* diverge. For example, if program P_1 can give output 1 or diverge on any of its inputs, then it will be identified with the program Q which diverges everywhere, since $\{\!|1,\perp|\!\} = \perp = \{\!|\perp|\!\}$ in $(\mathsf{N}_\perp)^\sharp$. However, program a P_2 which always gives 1 as its output (on inputs other than $\perp$) will *not* have the same meaning as P_1 and $\lambda x.\ \perp$. On the other hand, if the lower powerdomain is used in the interpretation of these programs, then P_1 and P_2 will be given the same meaning since $\{\!|1,\perp|\!\} = \{\!|1|\!\}$ in $(\mathsf{N}_\perp)^\flat$. However, P_1 and P_2 will not have the same meaning as the always divergent program Q since $\{\!|1,\perp|\!\} \neq \perp$ in the lower powerdomain. Finally, in the convex powerdomain, *none* of the programs P_1, P_2, Q have the same meaning since $\{\!|1,\perp|\!\}$, $\{\!|1|\!\}$ and $\{\!|\perp|\!\}$ are all distinct in $(\mathsf{N}_\perp)^\natural$.

To derive properties of the powerdomains like those that we discussed in the previous section for the other operators, we need to introduce the concept of a domain with binary operator.

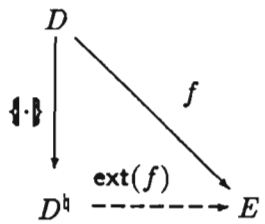
Definition: A *continuous algebra (of signature (2))* is a cpo E together with a continuous binary function $* : E \times E \to E$. We refer to the following collection of axioms on $*$ as theory $T^\natural$:

1. associativity: $(r * s) * t = r * (s * t)$
2. commutativity: $r * s = s * r$
3. idempotence: $s * s = s$.

(These are the well-known semi-lattice axioms.) A *homomorphism* between continuous algebras D and E is a continuous function $f : D \to E$ such that $f(s * t) = f(s) * f(t)$ for all $s, t \in D$. ∎

It is easy to check that, for any domain D, each of the algebras $D^\natural$, $D^\sharp$ and $D^\flat$ satisfies $T^\natural$. However, $D^\natural$ is the "free" continuous algebra over D which satisfies $T^\natural$:

Theorem 12 *Let D be a domain. Suppose $\langle E, * \rangle$ is a continuous algebra which satisfies $T^\natural$. For any continuous $f : D \to E$, there is a unique homomorphism $\mathsf{ext}(f) : D^\natural \to E$ which completes the following diagram:*



Proof: (Hint) If $u = \{x_1, \ldots, x_n\} \in s \in D^\natural$, and $\hat{u}$ is the principal ideal generated by u, then define $\mathbf{ext}(f)(\hat{u}) = f(x_1) * \cdots * f(x_n)$. This function has a unique continuous extension to all of $D^\natural$ given by $\mathbf{ext}(f)(s) = \bigsqcup\{\mathbf{ext}(f)(\hat{u}) \mid u \in s\}$. ∎

Now, consider the following axiom:

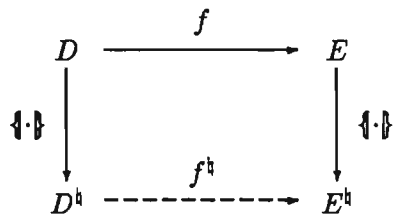
$4^\sharp$. $s \uplus t \sqsubseteq s$.

Let $T^\sharp$ be the set of axioms obtained by adding axiom $4^\sharp$ to the axioms in $T^\natural$. Similarly, let $T^\flat$ be obtained by adding the axiom

$4^\flat$. $s \sqsubseteq s \uplus t$

to the axioms in $T^\natural$. The point is this: *Theorem 12 still holds when $D^\natural$ and $T^\natural$ are replaced by $D^\sharp$ and $T^\sharp$ respectively,* or *by $D^\flat$ and $T^\flat$ respectively.*

As was the case with the smash product and lift operators, a diagram like the one in Theorem 12 gives rise to another important operation on functions. If $f : D \to E$ is a continuous function, then there is a unique homomorphism $f^\natural$ which completes the following diagram:



Namely, one defines $f^\natural = \mathbf{ext}(\{|\cdot|\} \circ f)$. Of course, there are functions $f^\sharp$ and $f^\flat$ with similar definitions.

Two of the powerdomains preserve the property of bounded completeness:

Lemma 13 *If D is a bounded complete domain then so are $D^\sharp$ and $D^\flat$.*

Proof: We leave for the reader the exercise of showing that a domain D is bounded complete if and only if every finite bounded subset of its basis has a least upper bound. To see that $D^\flat$ is bounded complete, just note that, for any pair of sets $u, v \in \mathcal{P}^*_f(K(D))$, the ideal generated by their union $u \cup v$ is the least upper bound in $D^\flat$ for the ideals generated by u and v. To see that $D^\sharp$ is bounded complete, suppose $u, v, w \in \mathcal{P}^*_f(K(D))$ with $w \vdash^\sharp u$ and $w \vdash^\sharp v$. Let w' be the set of elements $z \in K(D)$ such that there are elements $x \in u$ and $y \in v$ and z is the least upper bound of $\{x, y\}$. The set w' is non-empty because $\{u, v\}$ is bounded. Moreover, it is not hard to see that $w \vdash^\sharp w'$ and $w' \vdash^\sharp u$ and $w' \vdash^\sharp v$. Hence the ideal generated by w' is the least upper bound of the ideals generated by u and v. ∎

1.6 Bifinite domains.

Of the operators that we have discussed so far, only the convex powerdomain $(\cdot)^\natural$ does not take bounded complete domains to bounded complete domains. To see this in a simple example, consider the finite poset $\mathsf{T} \times \mathsf{T}$ and the following elements of $\mathcal{P}_f^*(\mathsf{T} \times \mathsf{T})$:

$$u = \{\langle \bot, \mathsf{true}\rangle,\ \langle \bot, \mathsf{false}\rangle\}$$
$$v = \{\langle \mathsf{true}, \bot\rangle,\ \langle \mathsf{false}, \bot\rangle\}$$
$$u' = \{\langle \mathsf{true}, \mathsf{true}\rangle,\ \langle \mathsf{false}, \mathsf{false}\rangle\}$$
$$v' = \{\langle \mathsf{true}, \mathsf{false}\rangle,\ \langle \mathsf{false}, \mathsf{true}\rangle\}$$

It is not hard to see that u' and v' are *minimal* upper bounds for $\{u, v\}$ with respect to the ordering $\sqsubseteq^\natural$. Hence no *least* upper bound for $\{u, u'\}$ exists and $(\mathsf{T} \times \mathsf{T})^\natural$ is therefore not bounded complete. In this section we introduce a natural class of domains on which *all* of the operators we have discussed above (including the convex powerdomain) are closed. This class is defined as follows:

Definition: Let D be a cpo. Let $\mathcal{M}$ be the set of finitary projections with finite image. Then D is said to be *bifinite* if $\mathcal{M}$ is countable, directed and $\bigsqcup \mathcal{M} = \mathsf{id}$. ∎

The bifinite cpo's are motivated, in part, by considerations from category theory and the definition above is a restatement of their categorical definition. They were first defined by Plotkin [Plo76] (where they are called "**SFP**-objects") and the term "bifinite" is due to Paul Taylor. Bifinite domains (and various closely related classes of cpo's) have also been discussed under other names such as "strongly algebraic" [Smy83a, Gun86] and "profinite" [Gun87] domains.

1.6.1 Plotkin orders.

As we suggested earlier, the image of a finitary projection $p : D \to D$ on a domain D can be viewed as an approximation to D. A bifinite domain is one which is a directed limit of its finite approximations. But what is this really saying about the structure of D? First of all, it follows from properties of finitary projections that we mentioned earlier that whenever $p : D \to D$ is a finitary projection and $\mathsf{im}(p)$ is finite, then $\mathsf{im}(p) \subseteq K(D)$. From this, together with the fact that the set $\mathcal{M}$ is directed and $\bigsqcup \mathcal{M} = \mathsf{id}$, it is possible to show D is a domain with $\bigcup\{\mathsf{im}(p) \mid p \in \mathcal{M}\}$ as its basis. We may now use the correspondence which we noted in Theorem 6 to provide a condition on the basis of a domain which characterizes the domain as being bifinite. Recall that $N \lhd A$ for posets N and A if $N \cap {\downarrow} x$ is directed for every $x \in A$.

Definition: A poset A is a *Plotkin order* if, for every finite subset $u \subseteq A$, there is a finite set $N \lhd A$ with $u \subseteq N$. ∎

Theorem 14 *The following are equivalent for any cpo D.*

1. *D is bifinite.*

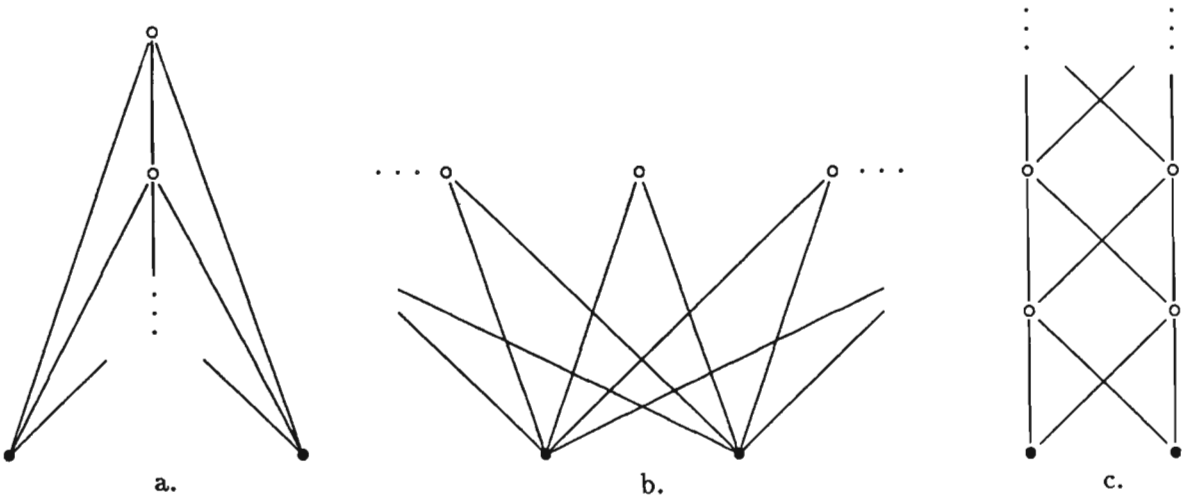


Figure 1.3: Posets that are not Plotkin orders.

2. D is a domain and $K(D)$ is a Plotkin order. ∎

To get some idea what a Plotkin order looks like, it helps to have a definition. Given a poset A and a finite set $u \subseteq A$, an upper bound x for u is *minimal* if, for any upper bound y for u, $y \sqsubseteq x$ implies $y = x$. A set v of minimal upper bounds for u is said to be *complete* if, for every upper bound x for u, there is a $y \in v$ with $y \sqsubseteq x$. Now, let A be a Plotkin order and suppose $u \subseteq A$ is finite. Then there is a finite $N \triangleleft A$ with $u \subseteq N$. The set N must contain a complete set of minimal upper bounds for u (why?). This shows the first fact about Plotkin orders: every finite subset has a complete set of minimal upper bounds. This rules out configurations like the one pictured in Figure 1.3a where the pair of points indicated by closed circles do not have such a complete set of minimal upper bounds. But the set N is *finite* so we have our second fact: every finite subset must have a *finite* complete set of minimal upper bounds. This rules out configurations like the one pictured in Figure 1.3b where the pair of points indicated by closed circles has a complete set of minimal upper bounds but not a finite one. However, having finite complete sets of minimal upper bounds for finite subsets is not a sufficient condition for characterizing the Plotkin orders. To see why, let A be a poset which has finite complete sets of minimal upper bounds for finite subsets. If $u \subseteq A$ is finite, let

$$\mathcal{U}(u) = \{x \mid x \text{ is the minimal upper bound for some } v \subseteq u\}.$$

Now, if $u \subseteq N \triangleleft A$, then $\mathcal{U}(u) \subseteq N$. Hence, $\mathcal{U}^n(u) \subseteq N$ for each n. If N is finite, then there must be an n for which $\mathcal{U}^n(u) = \mathcal{U}^{n+1}(u)$. This is a third fact about Plotkin orders: for each finite $u \subseteq A$, $\mathcal{U}^\infty(u) = \bigcup_n \mathcal{U}^n(u)$ is finite. To see what can go wrong, note that $\mathcal{U}^\infty(u)$ is infinite when u is the pair of points indicated by closed circles in Figure 1.3c.

1.6.2 Closure properties.

Proposition 15 *A bounded complete domain is bifinite.*

Proof: Suppose D is bounded complete and $u \subseteq K(D)$ is a finite subset of the basis of D. Let

$$N = \{x \mid x \text{ is the least upper bound of a finite subset of } u\ \}.$$

Note that N is finite; we claim that $N \triangleleft K(D)$. Suppose x is the least upper bound of a finite set $v \subseteq K(D)$. Since D is algebraic, there is a directed subset $M \subseteq K(D)$ such that $x = \bigsqcup M$. But the elements of v are compact. Hence, for every $y \in v$, there is a $y' \in M$ with $y \sqsubseteq y'$. Since M is directed, there is some $z \in M$ which is an upper bound for v. Now, $z \sqsubseteq x$ so $x = z$ and x is therefore compact. This shows that $N \subseteq K(D)$. Suppose $v \subseteq N$ is bounded, then the least upper bound of v is the same as the least upper bound of the set $\{x \in u \mid x \sqsubseteq y \text{ for some } y \in v\}$ so the least upper bound of v is in N. Now, if $x \in K(D)$, then $S = (\downarrow x) \cap N$ is bounded. Since S has a least upper bound which, apparently, lies in S, we conclude that S is directed. ∎

Theorem 16 *If D is bifinite, then the poset $\mathsf{Fp}(D)$ of finitary projections on D is an algebraic lattice and the inclusion map $i : \mathsf{Fp}(D) \hookrightarrow (D \to D)$ is an embedding.*

Proof: (Sketch) One uses Theorem 6 to show that $\mathsf{Fp}(D)$ is an algebraic lattice. Suppose $f : D \to D$ is continuous. Let

$$S_f = \{x \in K(D) \mid x \sqsubseteq f(x)\}.$$

One can show that there is a least set N_f such that $S_f \subseteq N_f \triangleleft K(D)$. This set determines a finitary projection p_{N_f} as in the discussion before Theorem 6. On the other hand, if $f : D \to D$ is a finitary projection then $N_f = \mathsf{im}(f) \cap K(D)$ and $f = p_{N_f}$. The remaining steps required to verify that $f \mapsto N_f$ is a projection are straight-forward. ∎

Lemma 17 *If D and E are bifinite domains, then so are the cpo's $D \to E$, $D \multimap E$, $D \times E$, $D \otimes E$, $D + E$, $D \oplus E$, $D_\perp$, $D^\natural$, $D^\sharp$ and $D^\flat$.*

Proof: We will outline proofs for two sample cases. We begin with the function space operator. Suppose $p : D \to D$ and $q : E \to E$ are finitary projections. Given a continuous function $f : D \to E$, define $\Theta(q,p)(f) = q \circ f \circ p$. The function $\Theta(q,p)$ defines a finitary projection on $D \to E$. Moreover, if p and q have finite images, then so does $\Theta(q,p)$. If we let $\mathcal{M}$ be the set of functions $\Theta(q,p)$ such that p and q are finitary projections with finite image, then it is easy to see that $\bigsqcup \mathcal{M} = \mathsf{id}$. Hence $D \to E$ is bifinite. We will encounter the function Θ again in the next section.

To see that $D^\natural$ is bifinite, one shows that the set

$$\mathcal{M} = \{p^\natural \mid p \in \mathsf{Fp}(D) \text{ and } \mathsf{im}(p) \text{ is finite}\}$$

is directed and has the identity as its least upper bound. The functions in $\mathcal{M}$ are themselves finitary projections with finite images so $D^\natural$ is bifinite. ∎

One may conclude from this lemma that the bifinite domains have rather robust closure properties. But there is something else about bifinite domains which makes them special. They are the *largest* class of domains which are closed under the operators listed in the Lemma. In fact, there is the following:

Theorem 18 *If D and $D \to D$ are domains, then D is bifinite.* ∎

The theorem is due to Smyth and its proof may be found in [Smy83a]. It is carried out by analyzing each of the cases pictured in Figure 1.3 and showing that if $D \to D$ is not a domain, then D cannot be bifinite. A similar result for the bounded complete domains can be found in [Gun86].

1.7 Recursive definitions of domains.

Many of the data types that arise in the semantics of computer programming languages may be seen as solutions of *recursive domain equations*. Consider, for example, the equation $T \cong T + T$ (of course, this is an *isomorphism* rather than an *equality*, but let us not make much of this distinction for the moment). How would we go about finding a domain which solves this equation? Suppose we start with the one point domain $T_0 = \mathsf{I}$ as the first approximation to the desired solution. Taking the proof of the Fixed Point Theorem as our guide, we build the domain $T_1 = T_0 + T_0 = \mathsf{I} + \mathsf{I}$ as the second approximation. Now, there is a unique embedding $e_0 : T_0 \to T_1$ so this gives a precise sense in which T_0 approximates T_1. The next approximation to our solution is the domain $T_2 = T_1 + T_1$ and again there is an embedding $e_1 = e_0 + e_0 : T_1 \to T_2$. If we continue along this path we build a sequence

$$T_0 \xrightarrow{e_0} T_1 \xrightarrow{e_1} T_2 \xrightarrow{e_2} \cdots$$

of approximations to the full simple binary tree. To get a domain, we must add limits for each of the branches. The resulting domain (*i.e.* the full simple binary tree with the limit points added) is, indeed, a "solution" of $T \cong T + T$. This is all very informal, however; how are we to make this idea mathematically *precise* and, at the same time, sufficiently *general?*

1.7.1 Solving domain equations with closures.

In this section we discuss a technique for solving recursive domain equations by relating domains to functions by the "image" map (im) and then using the ideas of the previous section to solve equations. There are two (closely related) ways of doing this which we will illustrate. The first of these is based on the following concept:

Definition: Let D and E be cpo's. A continuous function $r : D \to E$ is a *closure* if there is a continuous function $s : E \to D$ such that $r \circ s = \mathsf{id}$ and $s \circ r \sqsupseteq \mathsf{id}$. ∎

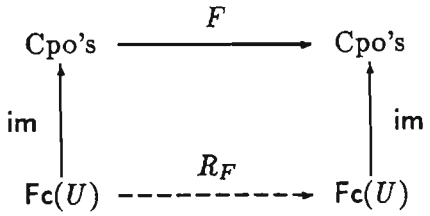
By analogy with the notion of a finitary projection, we will say that a function $r : D \to D$ is a *finitary closure* if $r \circ r = r \sqsupseteq \mathsf{id}$ and $\mathsf{im}(r)$ is a domain. In the event that D is a domain, the requirement that $\mathsf{im}(r)$ be a domain is unnecessary because we have the following:

Lemma 19 *If D is a domain and $r : D \to D$ satisfies the equation $r \circ r = r \sqsupseteq \mathsf{id}$, then $\mathsf{im}(r)$ is a domain.* ∎

The Lemma is proved by showing that $\{r(x) \mid x \in K(D)\}$ forms a basis for $\mathsf{im}(r)$. We will say that a domain E is a *closure of* D if it is isomorphic to $\mathsf{im}(r)$ for some finitary closure r on D. We let $\mathsf{Fc}(D)$ be the poset of finitary closures $r : D \to D$.

Lemma 20 *If D is a domain, then $\mathsf{Fc}(D)$ is a cpo.* ∎

Definition: Let us say that an operator F on cpo's is *representable* over a cpo U if and only if there is a continuous function R_F which completes the following diagram (up to isomorphism):



i.e. $\mathsf{im}(R_F(r)) \cong F(\mathsf{im}(r))$ for every closure r. ∎

This idea extends to multiary operators as well. For example, the function space operator $\cdot \to \cdot$ is representable over a cpo U if there is a continuous function

$$R : \mathsf{Fc}(U) \times \mathsf{Fc}(U) \to \mathsf{Fc}(U)$$

such that, for any $r, s \in \mathsf{Fc}(U)$,

$$\mathsf{im}(R(r,s)) \cong \mathsf{im}(r) \to \mathsf{im}(s)$$

A operator $\langle F_1, \ldots, F_n \rangle$ is defined to be representable if each of the operators F_i is. Note that a composition of representable operators is representable.

Theorem 21 *If an operator F is representable over a cpo U, then there is a domain D such that $D \cong F(D)$.*

Proof: Suppose R_F represents F. By the Fixed Point Theorem, there is an $r \in \mathsf{Fc}(U)$ such that $r = R_F(r)$. Thus $\mathsf{im}(r) = \mathsf{im}(R_F(r)) \cong F(\mathsf{im}(r))$ so $\mathsf{im}(r)$ is the desired domain. ∎

Now we know how to solve domain equations. For example, to solve $T \cong T + T$ we need to find a domain U and continuous function $f : U \to U$ which represents the operator $F(X) = X + X$. But we are still left with the problem of finding a domain over which such operations may be represented! The next step is to look at a simple structure which can be used to represent several of the operations in which we are interested.

Given sets S and T, let T^S be the set of (all) functions from S into T. If T is a cpo, then T^S is also a cpo under the pointwise ordering. Now, it is not hard to see that the domain equation $X \cong X \times \mathsf{I}^\top$ (where $\mathsf{I}^\top$ is the two point lattice) has, as one of its solutions, the cpo $(\mathsf{I}^\top)^{\mathsf{N}}$. In fact, this cpo is isomorphic to the algebraic cpo $\mathcal{P}\mathsf{N}$ of subsets of N which we discussed in the first section. It is particularly interesting because of the following:

Theorem 22 *For any (countably based) algebraic lattice L, there is a closure $r : \mathcal{P}\mathsf{N} \to L$.*

Proof: Let $l_0, l_1, l_2, \ldots$ be an enumeration of the basis of L. Given $S \subseteq \mathsf{N}$, let $r(S) = \bigsqcup\{l_n \mid n \in S\}$. If $l \in L$, let $s(l) = \{n \mid l_n \sqsubseteq l\}$. We leave for the reader the (easy) demonstration that r, s are continuous with $r \circ s = \mathsf{id}$ and $s \circ r \sqsupseteq \mathsf{id}$. ∎

Structures such as $\mathcal{P}\mathsf{N}$ are often referred to as *universal domains* because they have a rich collection of domains as retracts. In the remainder of this section we will discuss two more similar constructions and show how they may be used to provide representations for operators.

Unfortunately, there is no representation for the operator $F(X) = X + X$ over $\mathcal{P}\mathsf{N}$. However, there are some much more interesting operators which *are* representable over $\mathcal{P}\mathsf{N}$. In particular,

Lemma 23 *The function space operator is representable over* $\mathcal{P}\mathsf{N}$.

Proof: Consider the algebraic lattice of functions $\mathcal{P}\mathsf{N} \to \mathcal{P}\mathsf{N}$. By Theorem 22, we know that there are continuous functions

$$\Phi_{\to} : \mathcal{P}\mathsf{N} \to (\mathcal{P}\mathsf{N} \to \mathcal{P}\mathsf{N})$$
$$\Psi_{\to} : (\mathcal{P}\mathsf{N} \to \mathcal{P}\mathsf{N}) \to \mathcal{P}\mathsf{N}$$

such that $\Phi_{\to} \circ \Psi_{\to} = \mathsf{id}$ and $\Psi_{\to} \circ \Phi_{\to} \sqsupseteq \mathsf{id}$. Now, suppose $r, s \in \mathsf{Fc}(\mathcal{P}\mathsf{N})$ (that is, $r \circ r = r \sqsupseteq \mathsf{id}$ and $s \circ s = s \sqsupseteq \mathsf{id}$). Given a continuous function $f : \mathcal{P}\mathsf{N} \to \mathcal{P}\mathsf{N}$, let $\Theta(s,r)(f) = s \circ f \circ r$ and define

$$R_{\to}(r,s) = \Psi_{\to} \circ \Theta(s,r) \circ \Phi_{\to}.$$

To see that this function is a finitary closure, we take $x \in \mathcal{P}\mathsf{N}$ and compute

$$\begin{aligned}
& (R_{\to}(r,s) \circ R_{\to}(r,s))(x) \\
= \; & (\Psi_{\to} \circ \Theta(s,r) \circ \Phi_{\to})(\Psi_{\to}(s \circ (\Phi_{\to}(x)) \circ r) \\
= \; & (\Psi_{\to} \circ \Theta(s,r) \circ \Phi_{\to} \circ \Psi_{\to})(s \circ (\Phi_{\to}(x)) \circ r) \\
= \; & (\Phi_{\to} \circ \Theta(s,r))(s \circ (\Phi_{\to}(x)) \circ r) \\
= \; & \Psi_{\to}((s \circ s) \circ (\Phi_{\to}(x)) \circ (r \circ r)) \\
= \; & \Psi_{\to}(s \circ (\Phi_{\to}(x)) \circ r) \\
= \; & R_{\to}(r,s)(x)
\end{aligned}$$

and

$$R_{\to}(r,s)(x) = \Psi_{\to}(s \circ (\Phi_{\to}(x)) \circ r) \sqsupseteq \Psi_{\to}(\Phi_{\to}(x)) \sqsupseteq x.$$

Thus we have defined a function,

$$R_{\to} : \mathsf{Fc}(\mathcal{P}\mathsf{N}) \times \mathsf{Fc}(\mathcal{P}\mathsf{N}) \to \mathsf{Fc}(\mathcal{P}\mathsf{N})$$

which we now demonstrate to be a representation of the function space operator.

Given $r, s \in \mathsf{Fc}(\mathcal{P}\mathsf{N})$, we must show that there is an isomorphism

$$\mathsf{im}(R(r,s)) \cong \mathsf{im}(r) \to \mathsf{im}(s)$$

for each $r, s \in \mathsf{Fc}(\mathcal{P}\mathsf{N})$. Now, there is an evident isomorphism between continuous functions $f : \mathsf{im}(r) \to \mathsf{im}(s)$ and continuous functions $g : \mathcal{P}\mathsf{N} \to \mathcal{P}\mathsf{N}$ such that $g = s \circ g \circ r$. We claim that $\Psi_{\to}$ cuts down to an isomorphism between such functions and the sets in the image of $R_{\to}(r,s)$. Since $\Phi_{\to} \circ \Psi_{\to} = \mathsf{id}$, we need only show that $(\Psi_{\to} \circ \Phi_{\to})(x) = x$ for each $x = R_{\to}(r,s)(x)$. But if

$$x = \Psi_{\to}(s \circ (\Phi_{\to}(x)) \circ r)$$

then

$$\begin{aligned}(\Psi_{\to} \circ \Phi_{\to})(x) &= (\Psi_{\to} \circ \Phi_{\to} \circ \Psi_{\to})(s \circ (\Phi_{\to}(x)) \circ r) \\ &= \Psi_{\to}(s \circ (\Phi_{\to}(x)) \circ r) \\ &= x\end{aligned}$$

Hence $\mathsf{im}(R_{\to}(r,s)) \cong \mathsf{im}(r) \to \mathsf{im}(s)$ and we may conclude that $R_{\to}$ represents $\to$ over $\mathcal{P}\mathsf{N}$. ∎

A similar construction can be carried out for the product operator. Suppose

$$\Phi_{\times} : \mathcal{P}\mathsf{N} \to (\mathcal{P}\mathsf{N} \times \mathcal{P}\mathsf{N})$$
$$\Psi_{\times} : (\mathcal{P}\mathsf{N} \times \mathcal{P}\mathsf{N}) \to \mathcal{P}\mathsf{N}$$

such that $\Phi_{\times} \circ \Psi_{\times} = \mathsf{id}$ and $\Psi_{\times} \circ \Phi_{\times} \sqsupseteq \mathsf{id}$. For $r, s \in \mathsf{Fp}(\mathcal{P}\mathsf{N})$ define

$$R_{\times}(r,s) = \Psi_{\times} \circ (r \times s) \circ \Phi_{\times}$$

We leave for the reader the demonstration that this makes sense and $R_{\times}$ represents the product operator.

Suppose that L is an algebraic lattice. Then there are continuous functions

$$\Phi_L : \mathcal{P}\mathsf{N} \to \mathcal{P}\mathsf{N}$$
$$\Psi_L : \mathcal{P}\mathsf{N} \to \mathcal{P}\mathsf{N}$$

such that $\Phi_L \circ \Psi_L = \mathsf{id}$ and $\Psi_L \circ \Phi_L \sqsupseteq \mathsf{id}$. Then the function

$$R_L(r,s) = \Psi_L \circ \Phi_L$$

represents the constant operator $X \mapsto L$ because $\mathsf{im}(\Psi_L \circ \Phi_L) \cong L$. A similar argument can be used to show that a constant operator $X \mapsto D$ is representable over a domain U if and only if D is a closure of U.

1.7.2 Modelling the untyped λ-calculus.

It is tempting to try to solve the domain equation $D \cong D \to D$ by the methods just discussed. Unfortunately, the equation $\mathsf{I} \cong \mathsf{I} \to \mathsf{I}$ (corresponding to the fact that on a one-point set there is only one possible self-map) shows that there is no guarantee that the result will be at all interesting. There has to be a way to build in some nontrival structure that is not wiped out by the fixed-point process. Methods are described in [Sco76a, Sco80a], but the following, from [Sco76b, Sco80b], is more direct and more general.

Lemma 24 *Let U be a non-trivial cpo. If the product and function space operators can be represented over U, then there are non-trivial domains D and E such that $E \cong E \times E$ and $D \cong D \to E$.*

Proof: We can represent $F(X) = U \times X \times X$ over U, so there is a closure A of U such that $A \cong U \times A \times A$. Thus $U \times A \cong U \times (U \times A \times A) \cong (U \times A) \times (U \times A)$. So $E = U \times A$ is non-trivial and $E \cong E \times E$. Now, E is a closure of U so $G(X) = X \to E$ is representable over U. Hence there is a cpo $D \cong D \to E$. This cpo is non-trivial because E is. ∎

Theorem 25 *If U is a non-trivial domain which represents products and function spaces, then there is a non-trivial domain D such that $D \cong D \times D \cong D \to D$ and D is the image of a closure on U.*

Proof: Let D and E be the domains given by Lemma 24. Then

$$D \times D \cong (D \to E) \times (D \to E) \cong D \to (E \times E) \cong D \to E \cong D$$

and

$$D \to D \cong D \to (D \to E) \cong (D \times D) \to E \cong D \to E \cong D. \blacksquare$$

We note, in fact, that D will have $\mathcal{P}\mathsf{N}$ itself represented by a closure on U. Hence, to get a non-trivial solution for $D \cong D \to D \cong D \times D$, take U in the theorem to be $\mathcal{P}\mathsf{N}$. What good is such a domain? The answer is that a D satisfying these isomorphisms is a model for a very strong λ-calculus. If we expand the syntax of λ-calculus given in Section 5.3 of the chapter by Mosses to allow pairings, we would have:

$$E \ ::= \ (\lambda x.\ E) \mid E_1(E_2) \mid x \mid \mathsf{pair} \mid \mathsf{fst} \mid \mathsf{snd}$$

Now, Mosses points out that under the semantic function he defines, many *different* expressions are mapped into the *same* values. We can say that the model *satisfies* certain equations. In particular, under the isomorphisms obtained in our theorems above, the following equations will be satisfied:

1. $(\lambda x.\ E) = (\lambda y.\ [y/x]E)$ (provided y is not free in E)
2. $(\lambda x.\ E)(E') = [E'/x]E$
3. $(\lambda x.\ E(x)) = E$ (provided x is not free in E)
4. $\mathsf{fst}(\mathsf{pair}(E)(E')) = E$
5. $\mathsf{snd}(\mathsf{pair}(E)(E')) = E'$
6. $\mathsf{pair}(\mathsf{fst}(E))(\mathsf{snd}(E)) = E$

In these equations, the third and sixth especially emphasize the isomorphisms $D = D \to D$ and $D = D \times D$. There are models where $D \to D$ is represented by a closure on D (as is $D \times D$) but where this is not an isomorphism. It follows that the special equations are independent of the others.

In [Rev87] the question is brought up whether we can add to the above equations one relating functional abstraction with pairing. In particular, the following would be interesting:

$$\mathsf{pair}(x)(y) = (\lambda z.\ \mathsf{pair}(x(z))(y(z))).$$

This equation identifies the primitive pairing with what could be called *pointwise pairing*. This equation is independent from the others, but a model for it can be obtained from the first model by

introducing a new pairing and application operation that does things pointwise in a suitable sense. There must be many other kinds of models that relate the functional structure to other constructs as well.

Suppose we have domains that satisfy just the six equations. Then from the primitive operations given, many others can be defined. The operation of λ-abstraction is, to be sure, a variable-binding operator (somewhat like a quantifier), but the others are algebraic in nature. As stated, application is a binary operation, and **pair**, **fst** and **snd** are constants. But we can define binary, ternary, and unary operations such as: $\mathsf{pair}(x)(y)$, $\mathsf{pair}(x)(\mathsf{pair}(y)(z))$, $\mathsf{fst}(x)$, $\mathsf{snd}(y)$, $\mathsf{pair}(\mathsf{snd}(z))(\mathsf{fst}(z))$, and many, many more. In other words, the domain D will become a model of many kinds of algebras.

In general, an *algebra* is a set together with several operations defined on it, taking values in the same set. The simplest situation is to consider finitary operations (*i.e.*, operations taking a fixed finite number of arguments). When giving an algebra, the sequence of arities of the fundamental operations is called the *signature* of the algebra. Thus, a *ring* is often given with just two binary operations (*addition* and *multiplication*) making a signature (2,2). Now, subtraction is definable in first-order logic from addition, but the definition is not equational. Therefore, it may be better to consider a ring as an algebra of signature (2,2,2) with subtraction being taken as primitive. Of, course it is enough to have the minus operation, which is unary. So, a signature (2,1,2) is also popular. Strictly speaking, however, different signatures correspond to algebras of different types. Not every algebra of signature (2,2,2) is "equivalent" to one of signature (2,1,2); rings as algebras have very special properties.

By a *continuous algebra* we mean a domain with various continuous operations singled out. In particular, our λ-calculus model can be considered as a continuous algebra of signature (2,0,0,0,0,0). The binary operation is the operation of functional application. Here, 0 indicates a 0-ary operation, which is just a *constant*. We already know the constants **pair**, **fst**, **snd**. The other two popular constants from the literature on λ-calculus are called S and K. In terms of λ-abstraction they can be defined as follows:

$$S = (\lambda x.\ (\lambda y.\ (\lambda z.\ x(z)(y(z)))))$$
$$K = (\lambda x.\ (\lambda y.\ x))$$

They enjoy many, many equations in the algebra (see, for example, [Bar84]) and, in fact, any equation involving the λ-operator can be rewritten purely algebraically in terms of S and K and application.

We will call an expression in the notation of applicative algebra which has no variables a *combination*. Any combination F defines an n-ary operation:

$$F(x_1)(x_2)\cdots(x_n).$$

What we have been remarking is that the algebras so obtained from combinations can be very rich. In a series of papers [Eng81, Eng] Engeler discussed just how rich these algebras can be. A representative result, following Engeler, will be exhibited here.

Theorem 26 *Given a signature $(s_1, s_2, \ldots, s_n)$, there are combinations $F_1, F_2, \ldots, F_n$ defining operations on D of these arities such that whenever a continuous algebra of this signature is given*

on a domain A that is a retract of D, then A can be made isomorphic to a subalgebra of this fixed algebra structure on D.

Proof: If A is a retract of D, then A can be regarded as a subset of D, and all the continuous operations on A can be naturally extended to continuous operations on D of the same arities. (This does not solve the problem, since the operations on D depend on the choice of A. That is to say, at the start A is a subalgebra of the wrong algebra on D.) We can call these operations $o_1, o_2, \ldots, o_n$.

We are going to define the representation of A as a subalgebra of D by means of a continuous function $\rho : A \to D$ defined by means of a fixed-point equation:

$$\begin{aligned}
\rho(a) \;=\; & \mathsf{pair}(a) \\
& (\mathsf{pair}(\lambda x_2 \ldots \lambda x_{s_1}.\ \rho(o_1(a, \mathsf{fst}(x_2), \ldots, \mathsf{fst}(x_{s_1})))) \\
& (\mathsf{pair}(\lambda x_2 \ldots \lambda x_{s_2}.\ \rho(o_2(a, \mathsf{fst}(x_2), \ldots, \mathsf{fst}(x_{s_2})))) \\
& \quad \vdots \\
& (\mathsf{pair}(\lambda x_2 \ldots \lambda x_{s_n}.\ \rho(o_n(a, \mathsf{fst}(x_2), \ldots, \mathsf{fst}(x_{s_n})))) \\
& (K)\ )\cdots)
\end{aligned}$$

In this way, we build into ρ the elements from A and the operations as well. The question is how to read off the coded information.

Consider the following combinations:

$$\begin{aligned}
F_1 &= \lambda x.\ \mathsf{fst}(\mathsf{snd}(x)) \\
F_2 &= \lambda x.\ \mathsf{fst}(\mathsf{snd}(\mathsf{snd}(x))) \\
&\vdots \\
F_n &= \lambda x.\ \mathsf{fst}(\mathsf{snd}(\mathsf{snd}(\cdots \mathsf{snd}(x)))),
\end{aligned}$$

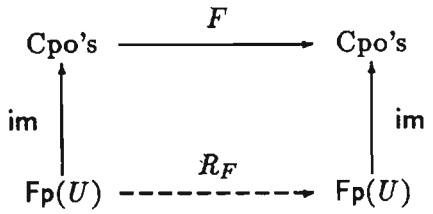
which have to be rewritten in terms of S, K, fst, and snd. We then calculate that

$$F_i(\rho(a_1))(\rho(a_2)) \cdots (\rho(a_{s_i})) = \rho(o_i(a_1, a_2, \ldots a_{s_i})).$$

This means if we consider the algebra $\langle D, F_1, F_2, \ldots, F_n \rangle$, then we can find by means of the definition of ρ any algebra $\langle A, o_1, o_2, \ldots, o_n \rangle$, isomorphic to a subalgebra of the first algebra. ∎

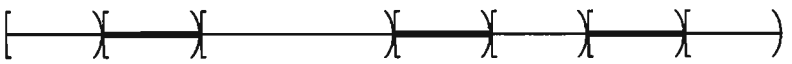
1.7.3 Solving domain equations with projections.

As we mentioned earlier, one slightly bothersome drawback to $\mathcal{P}\mathsf{N}$ as a domain for solving recursive domain equations is the fact that it cannot represent the sum operator $+$. One might try to overcome this problem by using the operator $(\ \cdot\ +\ \cdot\ )^{\top}$ as a substitute since this *is* representable over $\mathcal{P}\mathsf{N}$. However, the added top element seems unmotivated and gets in the way. It is probably possible to find a cpo which will represent the operators $\times, \to, +$. However, for the sake of variety, we will discuss a slightly different method for solving domain equations. Let us say that an operator F on cpo's is *p-representable* over a cpo U if and only if there is a continuous function R_F which completes the following diagram (up to isomorphism):



Since there will be no chance of confusion, let us just use the term "representable" for "p-representable" for the remainder of this section. Since $\mathsf{Fp}(U)$ is a cpo we can solve domain equations in the same way we did before *provided we can find domains over which the necessary operators can be represented.*

The construction of a suitable domain is somewhat more involved than was the case for $\mathcal{P}\mathsf{N}$. We begin by describing the basis of a domain U. Let S be the set of rational numbers of the form $n/2^m$ where $0 \leq n < 2^m$ and $0 < m$. As the basis U_0 of our domain we take finite (non-empty) unions of half open intervals $[r,t) = \{s \in S \mid r \leq s < t\}$. A typical element would look like



We order these sets by superset so that the interval $[0,1)$ is the *least* element. There is no top element under this ordering. If we adjoin the emptyset, say $B = \mathsf{U}_0 \cup \{\emptyset\}$, then we get a *Boolean algebra.* (Note that the complement of a finite union of intervals is again one such—unless it is empty.) In particular, any interval contains a proper sub-interval so, as a Boolean algebra, B is *atomless.* But B is countable, and—up to isomorphism—the only countable atomless Boolean algebra is the free one on countably many generators. But this Boolean algebra has the property that every countable Boolean algebra is isomorphic to a subalgebra. Now, suppose A is a countable bounded complete poset. Let B' be the boolean algebra of subsets of A generated by those subsets of the form $\uparrow x = \{y \in A \mid x \sqsubseteq y\}$ and order this collection by superset so that $\emptyset$ will be its largest element. The map $i : x \mapsto \uparrow x$ is a monotone injection which preserves existing least upper bounds. Moreover, a subset $u \subseteq A$ is bounded just in case $\bigcap_{x \in u} \uparrow x$ is non-empty. Now, if $j : B' \to B$ maps B' isomorphically onto a subalgebra of B, then the composition $j \circ i$ cuts down to an isomorphism between A and a normal subposet $A' \lhd \mathsf{U}_0$. Letting U be the domain of ideals over U_0 we may now conclude the following:

Theorem 27 *For any bounded complete domain D, there is a projection*

$$p : \mathsf{U} \to D. \ \blacksquare$$

We can now use this to see that an equation like $X \cong \mathsf{N}_\perp + (X \to X)$ has a solution. The proof that $\to$ is representable over U is almost identical to the proof we gave above that it is representable over $\mathcal{P}\mathsf{N}$. To get a representation for $+$, take a pair of continuous functions

$$\Phi_+ : \mathsf{U} \to (\mathsf{U} + \mathsf{U})$$
$$\Psi_+ : (\mathsf{U} + \mathsf{U}) \to \mathsf{U}$$

such that $\Phi_+ \circ \Psi_+ = \mathsf{id}$ and $\Psi_+ \circ \Phi_+ \sqsubseteq \mathsf{id}$. Then take

$$R_+(r,s) = \Psi_+ \circ (r+s) \circ \Phi_+.$$

Also, there is a representation $R_{\mathsf{N}_\perp}$ for constant operator $X \mapsto \mathsf{N}_\perp$. Hence the operator $X \mapsto \mathsf{N}_\perp + (X \to X)$ is represented over U by the function

$$p \mapsto R_+(R_{\mathsf{N}_\perp}(p), R_\to(p,p)).$$

We have, in fact, the following:

Lemma 28 *The following operators are representable over* U*:* $\to$, $\multimap$, $\times$, $\otimes$, $+$, $\oplus$, $(\cdot)_\perp$, $(\cdot)^\sharp$, $(\cdot)^\flat$. ∎

This means that we have solutions over the bounded complete domains for a quite substantial class of recursive equations. More discussion of U may be found in [Sco81], [Sco82a] and [Sco82b].

1.7.4 Representing operators on bifinite domains.

The convex powerdomain $(\cdot)^\natural$ cannot be representable over U because it does not preserve bounded completeness. We construct a domain over which this operator can be represented as follows. Given a poset A, define $M(A)$ to be the of pairs $(x,u) \in A \times \mathcal{P}_f(A)$ such that $x \sqsubseteq z$ for every $z \in u$. Define a pre-ordering on $M(A)$ by setting $(x,u) \vdash (y,v)$ if and only if there is a $z \in u$ such that $z \sqsubseteq y$. Now, given a domain D, we define D^+ to be the domain of ideals over $\langle M(A), \vdash \rangle$.

Theorem 29 *If D is bifinite, then so is D^+. Moreover, if $D \cong D^+$ and E is any bifinite domain, then there is a projection $p : D \to E$.* ∎

A full proof of the theorem may be found in [Gun87]. We will attempt to offer some hint about how the desired fixed point is obtained. At the first step we take the domain $\mathsf{I} = \{\perp\}$ containing only the single point $\perp$. At the second step, I^+, there are elements $a = (\perp, \{\perp\})$ and $b = (\perp, \emptyset)$ with $b \vdash a$. At the third step there are five elements

$$(a,\{a\}),\ (a,\{b\}),\ (b,\{b\}),\ (b,\emptyset),\ (a,\emptyset)$$

which form the partially ordered set I^{++} pictured in Figure 1.4. Note that there is another element $(a,\{a,b\}) \in M(\mathsf{I}^+)$ but this satisfies $(a,\{a\}) \vdash (a,\{a,b\})$ and $(a,\{a,b\}) \vdash (a,\{a\})$ so we have identified these elements in the picture. The next step I^{+++} has 20 elements (up to equivalence in the sense just mentioned) and it is also pictured in Figure 1.4. We leave the task of drawing a picture of I^{++++} as an exercise for the (zealous) reader. It should be noted that each stage of the construction is *embedded* in the next one by the map $x \mapsto (x,\{x\})$. The closed circles in the figure are intended to give a hint of how this embedding looks.

The technique which we have used to build this domain can be generalized and used for other classes as well [GJ88].

We have the following:

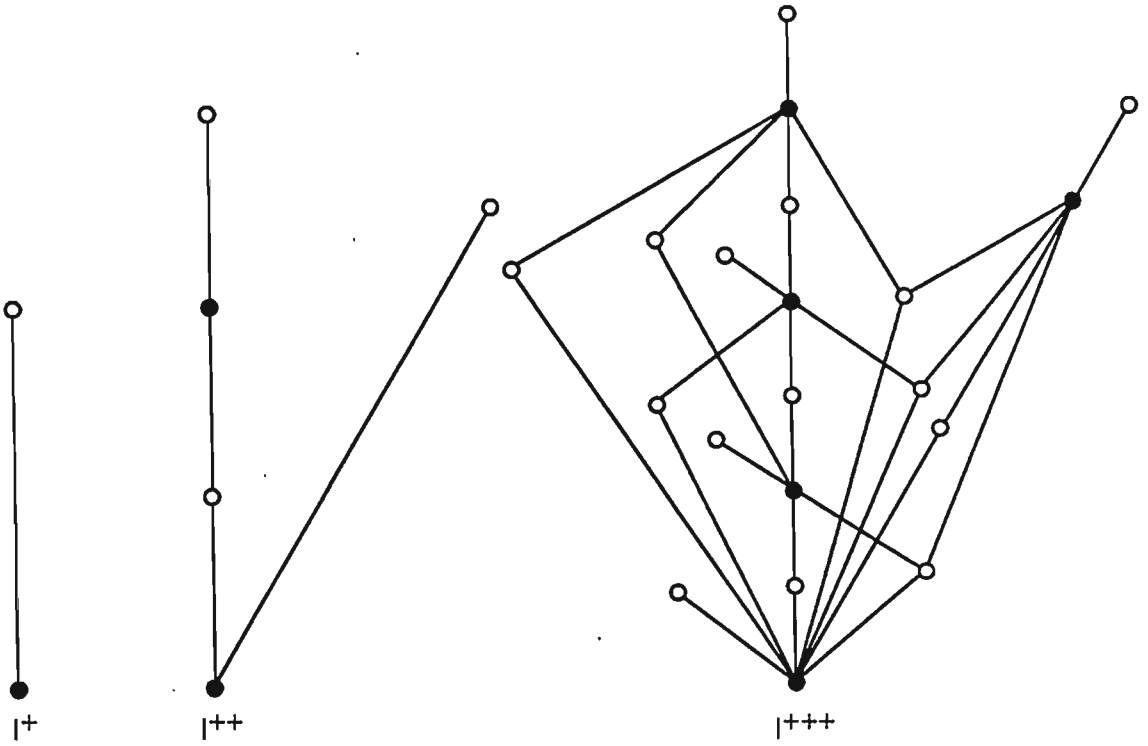


Figure 1.4: A domain for representing operators on bifinites.

Lemma 30 *The following operators are p-representable over* V: $\rightarrow$, $\multimap$, $\times$, $\otimes$, $+$, $\oplus$, $(\cdot)_\perp$, $(\cdot)^\sharp$, $(\cdot)^\flat$, $(\cdot)^\natural$. ∎

As with most of the other operators, to get a representation for $(\cdot)^\natural$, take a pair of continuous functions

$$\Phi_\natural : \mathsf{V} \to \mathsf{V}^\natural$$
$$\Psi_\natural : \mathsf{V}^\natural \to \mathsf{V}$$

such that $\Phi_\natural \circ \Psi_\natural = \mathsf{id}$ and $\Psi_\natural \circ \Phi_\natural \sqsubseteq \mathsf{id}$. Then

$$R_\natural(p) = \Psi_\natural \circ (p^\natural) \circ \Phi_\natural$$

is a representation for the convex powerdomain operator.

We hope that the reader has begun to note a pattern in the way operators are represented. Most of the operators $(\times, \otimes, +, \oplus, (\cdot)_\perp, (\cdot)^\sharp, (\cdot)^\flat, (\cdot)^\natural)$ may be handled rather straight-forwardly using the corresponding action of these operators on functions. Slightly more care must be taken in dealing with the function space and strict function space operators where one must use a function like Θ. The stock of operators that we have defined in this chapter is quite powerful and it can be used for a wide range of denotational specifications. However, the methods that we have used to show facts such as representability (using finitary closures or finitary projections) will apply to a very large class of operators which satisfy certain sufficient conditions.

To understand this phenomenon, one must pass to a more general theory in which such operators are a basic topic of study. This is the theory of *categories*. Many people find it difficult to gain access to the theory of domains when it is described with categorical terminology. On the other hand, it is difficult to explain basic concepts of domain theory without the extremely useful general language of category theory. A good exposition of the relevance of category theory to the theory of semantic domains may be found in [SP82].

Only a small number of categories of spaces having the properties which we have described above are known to exist. What are the special traits that these categories possess? First of all, they have product and function space functors which satisfy the relationship we described at the beginning of section 4. This property, known as *cartesian closure* is a well-known characteristic of categories such as that of sets and functions. But our cartesian closed categories have not only fixed points for (all) morphisms but fixed points for many functors as well. It is this latter feature which makes them well adapted to the task of acting as classes of semantic domains. One additional property which makes these categories special is the existence of domains for representing functors.

This is not to say that there are not other categories which will have the desired properties. One particularly interesting example are the stable structures of Berry [Ber78] which we have not had the space to discuss here. Interesting new examples of such categories are being uncovered by researchers at the time of the writing of this chapter. The reader will find a few leads to such examples in the published literature listed below, and we expect that many quite different approaches will be put forward in future years.

Bibliography

[Bar84] H. Barendregt. *The Lambda Calculus: Its syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, revised edition, 1984.

[Ber78] Gèrard Berry. Stable models of typed λ-calculus. In *International Colloquium on Automata, Languages and Programs*, pages 72–89, *Lecture Notes in Computer Science vol. 62,* Springer, 1978.

[Eng] E. Engeler. A combinatory representation of varieties and universal classes. *Algebra Universalis*, To appear.

[Eng81] E. Engeler. Algebra and combinators. *Algebra Universalis*, 13:389–392, 1981.

[GHK*80] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices.* Springer, 1980.

[GJ88] C. A. Gunter and A. Jung. Coherence and consistency in domains (extended outline). In Y. Gurevich, editor, *Logic in Computer Science*, pages 309–319, IEEE Computer Society, July 1988.

[Gun86] C. A. Gunter. The largest first-order axiomatizable cartesian closed category of domains. In A. Meyer, editor, *Logic in Computer Science*, pages 142–148, IEEE Computer Society, June 1986.

[Gun87] C. A. Gunter. Universal profinite domains. *Information and Computation*, 72:1–30, 1987.

[KT84] T. Kamimura and A. Tang. Effectively given spaces. *Theoretical Computer Science*, 29:155–166, 1984.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5:452–487, 1976.

[Rev87] G. Révésv. Rule-based semantics for an extended lambda-calculus. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, pages 43–56, *Lecture Notes in Computer Science vol. 298,* Springer, April 1987.

[Sco76a] D. S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5:522–587, 1976.

[Sco76b] D. S. Scott. Logic and programming languages. *Communications of the ACM*, 20:634–641, 1976.

[Sco80a] D. S. Scott. The lambda calculus: some models, some philosophy. In J. Barwise, editor, *The Kleene Symposium*, pages 381–421, North-Holland, 1980.

[Sco80b] D. S. Scott. Relating theories of the lambda calculus. In J. R. Hindley, editor, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 403–450, Academic Press, 1980.

[Sco81] D. S. Scott. Some ordered sets in computer science. In I. Rival, editor, *Ordered Sets*, pages 677–718, D. Reidel, 1981.

[Sco82a] D. S. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *International Colloquium on Automata, Languages and Programs*, pages 577–613, *Lecture Notes in Computer Science vol. 140,* Springer, 1982.

[Sco82b] D. S. Scott. Lectures on a mathematical theory of computation. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 145–292, *NATO Advanced Study Institutes Series,* D. Reidel, 1982.

[Smy77] M. Smyth. Effectively given domains. *Theoretical Computer Science*, 5:257–274, 1977.

[Smy78] M. Smyth. Power domains. *Journal of Computer System Sciences*, 16:23–36, 1978.

[Smy83a] M. Smyth. The largest cartesian closed category of domains. *Theoretical Computer Science*, 27:109–119, 1983.

[Smy83b] M. Smyth. Power domains and predicate transformers: a topological view. In J. Diaz, editor, *International Colloquium on Automata, Languages and Programs*, pages 662–676, *Lecture Notes in Computer Science vol. 154,* Springer, 1983.

[SP82] M. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11:761–783, 1982.

Chapter 2

Denotational Semantics.

2.1 Introduction

In programming linguistics, as in the study of natural languages, "syntax" is distinguished from "semantics". The *syntax* of a programming language is concerned only with the *structure* of programs: whether programs are "legal"; the connections and relations between the symbols and phrases that occur in them. *Semantics* deals with what legal programs *mean*: the "behaviour" they produce when executed by computers.

The topic of this chapter, Denotational Semantics, is a framework for the formal description of programming language semantics. The main idea of Denotational Semantics is that each phrase of the language described is given a *denotation*: a mathematical object that represents the contribution of the phrase to the meaning of any complete program in which it occurs. Moreover, the denotation of each phrase is determined just by the denotations of its subphrases.

Thus Denotational Semantics is concerned with giving mathematical *models* for programming languages. Models are *constructed* from given mathematical entities (functions, numbers, tuples, etc.). This is in contrast to the *axiomatic* approach used in other major frameworks, such as Hoare Logic [10] and Structured Operational Semantics [24].

The primary aim of Denotational Semantics is to allow *canonical definitions* of the meanings of programs. A canonical, denotational definition of a programming language documents the *design* of the language. It also establishes a *standard* for implementations of the language—ensuring that each program gives essentially the same results on all implementations that conform to the standard. A denotational definition does *not* specify the techniques to be used in implementations; it may, however, suggest some, and it has been shown feasible to develop implementations systematically from specifications written using the denotational approach. Finally, a denotational definition provides a basis for reasoning about the *correctness* of programs—either directly, or by means of derived proof rules for correctness assertions.

A further aim of Denotational Semantics is to promote *insight* regarding the concepts underlying programming languages. Such insight might help to guide the design of new (and perhaps "better") programming languages.

Currently, most programming language standards documents attempt to define semantics by means of *informal* explanations. This is in contrast to syntax, where formal grammars are routinely used in standards (in preference to informal explanations). However, experience has shown that informal explanations of semantics, even when they are carefully worded, are usually *incomplete* or *inconsistent* (or both), and open to misinterpretation by implementors. They are also an inadequate basis for reasoning about program correctness, and totally unsuitable for generation of implementations. These inherent defects of informal explanations do not afflict denotational definitions (except when definitions are left unfinished, or when their formal status is weakened by excessive use of informal abbreviations and conventions).

This chapter has two purposes. The first of these is to explain the *formalism* used in Denotational Semantics: *abstract syntax*, *semantic functions*, and *semantic domains*. Section 2.2 relates concrete syntax and abstract syntax. Section 2.3 considers the nature of semantic functions, and ex-

plains the properties of compositionality and full abstractness. Section 2.4 summarizes the concepts and notation of semantic domains, referring to Gunter and Scott [18] for a detailed presentation of domain theory.

The second purpose of this chapter is to illustrate the major standard *techniques* that are used in denotational descriptions of programming languages: *environments*, *stores*, *continuations*, etc. Section 2.5 explains the relation between these techniques and some fundamental concepts of programming languages, and uses the techniques to give denotational descriptions of many conventional programming constructs.

The Bibliographical Notes (Section 2.6) provide references to some significant works on Denotational Semantics.

The reader is expected to be familiar with the basic notions of discrete mathematics (sets, functions, relations, partial orders) and to be prepared to meet a substantial amount of formal notation. Familiarity with programming languages is an advantage, but not essential.

2.2 Syntax

As mentioned at the beginning, the *syntax* of a programming language is concerned only with the *structure* of programs: which programs are "legal"; what are the connections and relations between the symbols and phrases that occur in them.

There are several kinds of syntax, which we distinguish below. (Readers who are familiar with the distinction between "concrete syntax" and "abstract syntax" may prefer to skip to Section 2.2.3.)

2.2.1 Concrete syntax

Concrete Syntax treats a language as a set of *strings* over an alphabet of symbols.

Concrete syntax is usually specified by a grammar that gives "productions" for generating strings of symbols, using auxiliary "nonterminal" symbols. So-called "regular" grammars are inadequate for specifying syntax of programming languages: "context-free" grammars are required, at least.

Definition: A *context-free grammar* G is a quadruple (N, T, P, s_0) where N is a finite set of *nonterminal symbols*, T is a finite set of *terminal symbols* (disjoint from N), $P \subseteq N \times (N \cup T)^*$ is a finite set of *productions*, and $s_0 \in N$ is the *start symbol*.

(In this section, X^* is the set of strings over X, for any set X; the empty string is indicated by Λ, and string concatenation by juxtaposition. The notation X^* is given a different interpretation when X is a semantic domain, from Section 2.4 onwards.)

It is common practice to distinguish a *lexical* level and a *phrase* level in concrete syntax. The terminal symbols in the grammar specifying the lexical level are single characters; those in the phrase-level grammar are the *non*terminal symbols of the lexical grammar. Here, let us ignore the distinction between the lexical and phrase levels, for simplicity.

When presenting a grammar, it is enough to list the productions: the sets of nonterminal and terminal symbols are implicit, the start symbol is determined by the first production. We write a production $(a, (x_1 \ldots x_n))$ as $a ::= x_1 \ldots x_n$. We may also group several productions for the same nonterminal, separating the alternative strings on the right-hand side by '|'. (This notation for grammars is essentially the same as so-called *BNF*.) For later use, a mnemonic name called a *phrase sort* is associated with each nonterminal symbol (we write the phrase sort in parentheses) and occurrences of nonterminal symbols in right-hand sides of productions may be distinguished by subscripts.

An example of a grammar for the concrete syntax of a simple language of expressions is given in Table 2.1. (The productions for identifiers are omitted, as they are of no interest.)

We could define the language of strings generated by a context-free grammar in terms of "derivation steps". For our purposes here, it is more convenient to go straight to the notion of "derivation trees", in which the order of derivation steps is ignored.

(EXPRESSION)

E ::= T | E + T | E - T

(TERM)

T ::= F | T * F

(FACTOR)

F ::= I | (E)

(IDENTIFIER)

I ::= *unspecified*

Table 2.1: A grammar for concrete syntax

Definition: Let L be a set (of *labels*). An *L-labeled tree* t is a pair $(l,(t_1 \cdots t_n))$, where $l \in L$, $n \geq 0$, and $t_1 \ldots t_n$ is a string of L-labeled trees. We say that t has *label* l and *branches* $t_1,\ldots,t_n$.

Let Tree_L be the set of *finite* L-labeled trees; this is the *least* set that is closed under construction of L-labeled trees.

(Of course, other representations of trees are possible, e.g., as partial functions from "occurrences" to labels.)

Definition: For any $t \in \mathsf{Tree}_L$, $\mathsf{frontier}(t) \in L^*$ is defined (inductively) by

$$\mathsf{frontier}(l,(t_1 \cdots t_n)) = \begin{cases} l, & \text{if } n = 0; \\ \mathsf{frontier}(t_1) \cdots \mathsf{frontier}(t_n), & \text{if } n > 0. \end{cases}$$

Definition: A *derivation tree* according to a grammar $G = (N,T,P,s_0)$ is a finite $(N \cup T)$-labeled tree with label s_0, such that if a node labeled a has branches labeled $x_1,\ldots,x_n$, $n \geq 0$, then $(a,(x_1 \cdots x_n)) \in P$.

The set of all derivation trees according to G is denoted by Tree_G.

For notational convenience we identify the tree $(l,(\Lambda))$ with l. This allows us to write, e.g., $(a,(x_1\, t\, x_2))$ to form a derivation tree, where t is a tree and x_1, x_2 are terminal symbols. Figure 2.1 depicts a derivation tree according to the grammar of Table 2.1.

Now that we know what derivation trees are, let us use them to define the languages generated by grammars:

Definition: The *language of strings* $\mathcal{L}(G) \subseteq T^*$ generated by a grammar $G = (N,T,P,s_0)$ is given by

$$\mathcal{L}(G) = \{w \in T^* \mid \exists t \in \mathsf{Tree}_G : w = \mathsf{frontier}(t)\}.$$

If any string in $\mathcal{L}(G)$ has more than one derivation tree, then G is said to be *ambiguous*.

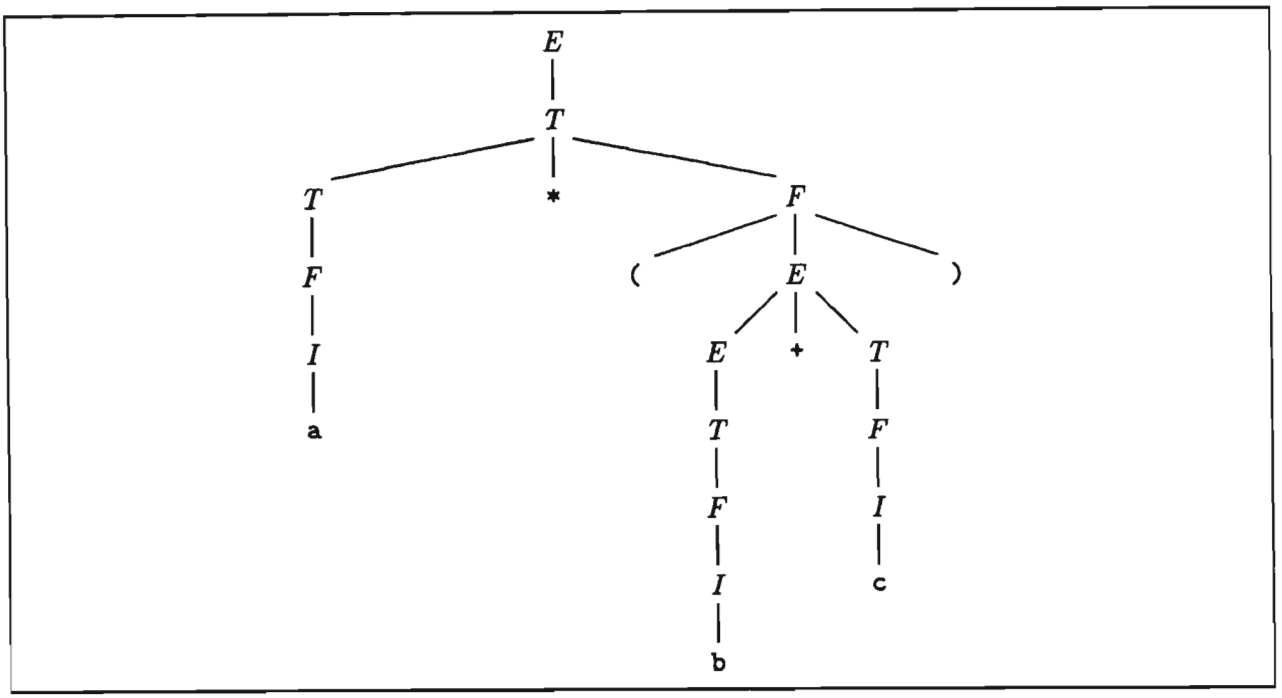


Figure 2.1: A derivation tree for concrete syntax

Whereas ambiguity seems to be an inescapable feature of natural languages, it is to be avoided in programming languages. For example, there should be no vagueness about whether '`a*b+c`' is to be read as '`a*(b+c)`' or as '`(a*b)+c`', since they should evaluate to different results, in general. (Of course grouping may not matter in some cases, such as '`a+b+c`'.) Moreover, the efficient generation of language parsers from grammars requires special kinds of unambiguous grammars, e.g., satisfying the so-called LALR(1) condition.

Unfortunately, unambiguous grammars tend to be substantially more complex than ambiguous grammars for the same language, and they often require nonterminal symbols and productions that have no relevance to the *essential* phrase structure of the language concerned. For the purposes of semantics, the phrase structure of languages should be as simple as possible, devoid of semantically-irrelevant details. Yet there should be no ambiguity in the structure of phrases! Thus we are led to use ambiguous grammars, but to interpret them in such a way that the specified syntactic entities themselves can be unambiguously decomposed. Such a framework is provided by so-called "abstract" syntax.

2.2.2 Abstract syntax

Abstract syntax treats a language as a set of *trees*. The important thing about trees is that, unlike strings, their compositional structure is *inherently* unambiguous: there is only one way of constructing a particular tree out of its (immediate) sub-trees.

It is convenient to use derivation trees to represent abstract syntax. Abstract syntax is specified using the same kind of (context-free) grammar that is used for concrete syntax—but now there is no

worry about ambiguity. An example of a grammar for abstract syntax is given in Table 2.2. It gives an appropriate abstract syntax for the language generated by the grammar of Table 2.1. Notice that the nonterminal symbols T and F, together with the terminal symbols '(' and ')', are not present in the abstract syntax: they were only for grouping in concrete syntax. Also, the various concrete expressions involving operators are collapsed into a single form of abstract expression—at the expense of introducing the nonterminal symbol O. Figure 2.2 shows a derivation tree according to the grammar of Table 2.2.

(EXPRESSION)

E ::= I | E_1 O E_2

(OPERATOR)

O ::= + | - | *

(IDENTIFIER)

I ::= *unspecified*

Table 2.2: A grammar for abstract syntax

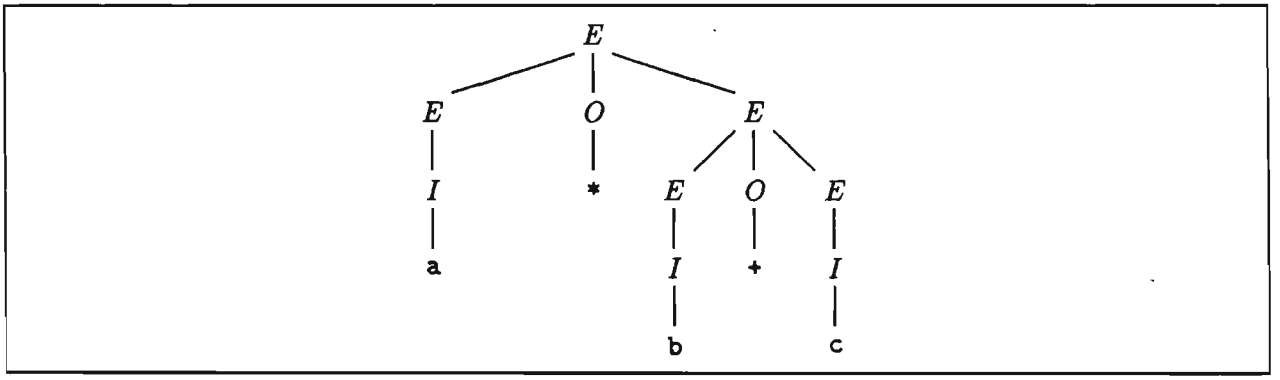


Figure 2.2: A derivation tree for abstract syntax

Definition: The *abstract syntax* defined by a grammar G is Tree_G, the set of derivation trees according to G.

The phrase sorts associated with nonterminal symbols in our grammars (such as EXPRESSION, IDENTIFIER) identify corresponding sets of derivation trees (note that such trees generally occur only as *branches* of trees in Tree_G).

Abstract syntax may be characterized *algebraically*, using the notion of a "signature", as follows.

Definition: Let S be a set (of *sorts*). An *S-sorted signature* Σ is a family of sets $\{\Sigma_{w,s}\}_{w\in S^*,s\in S}$ (of *operators*).

A *Σ-algebra* A consists of a family $\{A_s\}_{s\in S}$ of sets (called *carriers*) and for each operator $f \in \Sigma_{s_1\ldots s_n,s}$ a total function $f_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s$.

Definition: A Σ-*homomorphism* $h : A \to B$ (where A and B are Σ-algebras) is a family $\{h_s\}_{s \in S}$ of (total) functions $h_s : A_s \to B_s$ such that for each $f \in \Sigma_{s_1 \ldots s_n, s}$ and $a_i \in A_{s_i}$

$$f_B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) = h_s(f_A(a_1, \ldots, a_n)).$$

The composition $h' \circ h$ of Σ-homomorphisms $h : A \to B$, $h' : B \to C$ is the family of functions $\{h'_s \circ h_s\}_{s \in S}$. The identity Σ-homomorphism $\mathsf{id}_A : A \to A$ is the family of identity functions $\{\mathsf{id}_{A_s}\}_{s \in S}$. The Σ-algebras A, $B$are said to be *isomorphic* when there exist Σ-homomorphisms $h : A \to B$, $h' : B \to A$ such that $h' \circ h = \mathsf{id}_A$ and $h \circ h' = \mathsf{id}_B$.

The key concept is that of "initiality":

Definition: A Σ-algebra I is *initial* in a class $\mathbf{C}$ of Σ-algebras iff there is a unique Σ-homomorphism from I to each algebra in $\mathbf{C}$.

Proposition 31 *If I and J are both initial in a class $\mathbf{C}$ of Σ-algebras, then I and J are isomorphic.*

Proof: Let $h : I \to J$, $h' : J \to I$ be the unique homomorphisms given by the initiality of I, respectively J. Now $h' \circ h$ and id_I are both homomorphisms from I to itself; by the initiality of I they must be equal. Similarly, $h \circ h' = \mathsf{id}_J$.∎

For each grammar G we define a corresponding signature Σ_G, as follows:

Definition: Let $G = (N, T, P, s_0)$. Then Σ_G is the N-sorted signature with

$$\Sigma_{G, s_1 \ldots s_n, s} = \{p \in P \mid p = (s, (u_0 s_1 \ldots s_n u_n)); u_0, \ldots, u_n \in T^*\}$$

for each $(s_1 \ldots s_n) \in N^*$ and $s \in N$.

By the way, not all signatures can be made into context-free grammars: a signature may have an infinite number of sorts and operators. Notice also that a signature does not have a distinguished "start sort".

Now Tree_G can be made into a Σ_G-algebra, which we denote by $\mathcal{A}(G)$, as follows. Take the carriers $\mathcal{A}(G)_s$ to be $\mathcal{A}(N, T, P, s)$ for each $s \in N$. (In practice it is convenient to refer to these sets by mnemonic names, associated with nonterminal symbols when grammars are specified.) For each $p \in \Sigma_{G, s_1 \ldots s_n, s}$ with $p = (s, (u_0 s_1 \ldots s_n u_n))$, where $u_1, \ldots, u_n \in T^*$, define a function

$$p_{\mathcal{A}(G)} : \mathcal{A}(G)_{s_1} \times \cdots \times \mathcal{A}(G)_{s_n} \to \mathcal{A}(G)_s$$

by letting for all $t_i \in \mathcal{A}(G)_{s_i}$, for $i = 1, \ldots, n$,

$$p_{\mathcal{A}(G)}(t_1, \ldots, t_n) = (s, (u_0 t_1 \ldots t_n u_n)).$$

Proposition 32 *$\mathcal{A}(G)$ is initial in the class of all Σ_G-algebras.*

Proof: Let A be any Σ_G-algebra. Define $\{h_s : \mathcal{A}(G)_s \to A_s\}_{s \in S}$ inductively, as follows. If $t = (s, (u_0 t_1 \ldots t_n u_n))$ with each $u_i \in T^*$ and each $t_i \in \mathcal{A}(G)_{s_i}$, and $p = (s, (u_0 s_1 \ldots s_n u_n))$, then

$$h_s(t) = p_A(h_{s_1}(t_1), \ldots, h_{s_n}(t_n)).$$

As each $t \in \mathcal{A}(G)_s$ is uniquely decomposable as $(s, (u_0 t_1 \ldots t_n u_n))$, the h_s are well-defined (and they are total since derivation trees are finite). Moreover it can be seen that this definition is forced by the homomorphic property, so $\{h_s\}_{s \in S}$ is the unique homomorphism from $\mathcal{A}(G)$ to A. Hence $\mathcal{A}(G)$ is initial in the class of all Σ_G-algebras.∎

Denotational Semantics defines the semantics of a programming language on the basis of its abstract syntax. The semantics of some concrete syntax may be obtained as well: by giving a function that maps concrete syntax into abstract syntax. Assuming that the concrete grammar is unambiguous, it is enough to map concrete derivation trees into abstract derivation trees. (This map might be neither 1-1 nor onto, in general. Trying to invert it is called "pretty-printing", or "unparsing".)

The specification of the function from concrete to abstract syntax is quite trivial if the grammar for abstract syntax is obtained systematically from that for concrete syntax, just by "unifying" nonterminals and eliminating "chain productions". In fact the grammar of Table 2.2 was obtained from that of Table 2.1 (mainly) in that way; the corresponding map from concrete to abstract syntax may be imagined from the example where the tree in Figure 2.1 is mapped to that in Figure 2.2.

In general, it is up to the semanticist to *choose* an appropriate abstract syntax for a given language. Different choices may influence the difficulty of specifying the semantics. For instance, consider the rather trivial "language" of binary numerals, with concrete syntax given by the grammar in Table 2.3. Of course the semantics of binary numerals can be specified for the given syntax; but it turns out (as shown in Section 2.3) to be significantly simpler when the abstract syntax is given by the grammar in Table 2.4. (This latter grammar is unambiguous, so it could be used for concrete syntax as well as abstract syntax. But in general, the grammars used for abstract syntax are highly ambiguous, e.g., as for expressions in Table 2.2.)

(BINARY-NUMERAL)
B ::= 0 | 1 | 0 B | 1 B

Table 2.3: Concrete syntax for binary numerals

(BINARY-NUMERAL)
B ::= 0 | 1 | B 0 | B 1

Table 2.4: Abstract syntax for binary numerals

There do not seem to be any hard and fast rules for choosing grammars for abstract syntax. Usually, one has to compromise between on the one hand, keeping close to a given grammar for concrete syntax, and on the other hand, facilitating the semantic description.

Note that it is *not* required that the frontiers of the trees generated by the abstract grammar are the strings generated by the given concrete grammar, nor even that the same terminal symbols are used. In fact some authors prefer to use *disjoint* sets of symbols in concrete and abstract grammars, to avoid altogether any chance of confusion between concrete and abstract syntax. Here, we take the opposite position, and use symbols that make our grammars for abstract syntax strongly suggestive of familiar concrete syntax.

2.2.3 Context-sensitive Syntax

The grammars used here for specifying abstract syntax are context-free. But it is well-known that several features of programming languages are *context-sensitive*, and cannot be described by context-free grammars (e.g., that identifiers be declared before they are referred to, and that the "types" of operands match their operators).

In Denotational Semantics, context-sensitive syntax is regarded as a part of semantics, called *static semantics* (because it depends only on the program text, not on the input). For simplicity, let us assume that the static semantics of a program is just a truth-value indicating the legality of the program. Then the rest of the semantics of programs can be specified independently of their static semantics—the semantics of programs that are not legal (according to the static semantics) is defined, but irrelevant.

In practice, a proper treatment of static semantics might involve specification of error messages. Also, it may be convenient for a static semantics to yield abstract syntax that reflects context-sensitive disambiguations (for instance, whether occurrences of '+' are arithmetical, or set union), and to define the rest of the semantics on the disambiguated abstract syntax.

Static semantics is not considered further in this chapter. For a study of the semantics of types, see [26].

So much for syntax.

2.3 Semantics

Consider an entire program in some programming language. What is the nature of its semantics?

First of all let us dismiss any effects that the program might have on human readers, e.g., evoking feelings of admiration or (perhaps more often) disgust. In contrast to philology, programming linguistics is not concerned with subjective qualities at all. The semantics of a program is dependent only on the objective *behaviour* that the program causes (directly) when executed by computers.

Now computers are complex mechanisms, and all kinds of things can be observed to happen when they execute programs: lights flash, disc heads move, electric currents flow in circuits, characters appear on screens or on paper, etc. For programs that are specifically intended to control such physical behaviour, it would be necessary to consider these phenomena in their semantics.

But here, let us restrict our attention to programs whose behaviour is intended to be independent of particular computers. Such programs are typically written in general, high-level programming languages that actually deny the programmer direct control over the details of physical behaviour. The appropriate semantics of these programs is *implementation-independent*, consisting of just those features of program execution that are common to all implementations.

The implementation-independent semantics of a program may typically be modeled mathematically as a *function* (or relation) between *inputs* and *outputs*—where an input or output item might be just a number. The concrete representation of input and output as strings of bits is (usually) implementation-dependent, and hence ignored; likewise, the length of time taken for program execution. But *termination* properties are generally implementation-*in*dependent, and should therefore be taken into account in semantics.

Thus the semantics of a program is a mathematical object that models the program's implementation-independent behaviour. The semantics of a programming language consists of the semantics of all its programs.

Actually, some details of semantics are often left *implementation-defined*, e.g., limits on the size of numbers, maximum depth of recursive activation. These are regarded as *parameters* of the semantics; when such parameters are supplied, the implementation-independent semantics of a particular subclass of implementations is obtained.

A *standard* for implementations of a programming language may be established by:

(i) specifying the semantics of all programs in the language; and

(ii) specifying a "conformance" relation between semantic objects and implementation behaviours.

Our concern in this chapter is with (i), but let us digress for a moment to indicate how (ii) might be done.

Assume a correspondence between the inputs and outputs in the semantic model and some physical objects processed by implementations. Let a program and its input be given. If these uniquely determine output (and termination properties) then a conforming implementation, when given the physical representations of the program and input, must produce a representation of the

output—computing "for ever" if the semantics specifies non-termination. If, however, there are several possible outputs for a given program and input—i.e., the program is nondeterministic–the implementation need only produce one of them (perhaps not terminating if that is a possibility); the implementation may or may not be nondeterministic itself.

2.3.1 Denotations

Now back to our main concern: specifying the semantics of programs. The characteristic feature of Denotational Semantics is that one gives semantic objects for *all* phrases—not only for complete programs. The semantic object specified for a phrase is called the *denotation* of the phrase. The idea is that the denotation of each phrase represents the contribution of that phrase to the semantics of any complete program in which it may occur.

The denotations of compound phrases must depend only on the denotations of their subphrases. (Of course, the denotations of basic phrases do not depend on anything.) This is called *compositionality*.

It should be noted that the semantic analyst is free to *choose* the denotations of phrases—subject to compositionality. Sometimes there is a "natural", optimal choice, where phrases have the same denotations whenever they are *interchangeable* (without altering behaviour) in all *complete programs*; then the denotations are called *fully abstract*, and they capture just the "essential" semantics of phrases.

Note that considering interchangeability only in *complete programs* lets the notion of full abstractness refer directly to the *behaviours* of programs, rather than to their denotations. Different choices of which phrases are regarded as complete programs may give different conclusions concerning whether full abstractness obtains.

It is not always easy (or even possible) to find and specify fully abstract denotations, so in practice a compromise is made between simplicity and abstractness.

As an introductory (and quite trivial) example take the binary numerals. An abstract syntax for binary numerals was suggested in Section 2.2. Now let us extend the syntax to allow "programs" consisting of *signed* binary numerals, see Table 2.5.

(SIGNED-BINARY-NUMERAL)
Z ::= B | - B
(BINARY-NUMERAL)
B ::= 0 | 1 | B 0 | B 1

Table 2.5: Abstract syntax for signed binary numerals

The meanings (i.e., "behaviours") of signed binary numerals are supposed to be integers in $\mathbb{Z}$, according to the usual interpretation of binary notation (i.e., the most significant bit is the left-most), negated if preceded by '-'. We are free to choose the denotations for unsigned binary numerals B. The natural choice is to let each B denote the obvious natural number in $\mathbb{N}$, and such

denotations (specified formally in Section 2.3.2) are indeed fully abstract.

Any other choice of denotations is perhaps rather contrived in this simple example, but let us consider an alternative possibility so as to illustrate lack of full abstractness.

We could choose the denotation of B to be a pair $(n, l) \in \mathbf{N}^2$, where n gives the numerical value of B and l gives its *length*. Then the denotation of a signed binary numeral 'B' or '-B' would be determined just by n. Such denotations can be defined compositionally, but they are not fully abstract: for instance, the phrases '0' and '00' get distinct denotations, yet they can always be interchanged in any signed binary numeral without affecting its meaning.

Now consider the original (concrete) grammar for unsigned binary numerals (Table 2.4) and regard it as a specification of abstract syntax. With this phrase structure, we are no longer able to take the denotation of B to be just its numerical value: the value of the phrase '1B' is determined not only by the numerical value of B, but also by the number of its leading zeros. In fact the $(n, l) \in \mathbf{N}^2$ denotations mentioned above turn out to be fully abstract for *this* syntax.

The above example shows that the property of full abstractness can be rather sensitive to the structure of abstract syntax—and thereby casts doubt on its appropriateness as an absolute criterion of the quality of denotational descriptions.

In Denotational Semantics, there is in general a sharp distinction between syntax and semantics, and denotations consist of mathematical objects (such as numbers and functions) that exist completely independently of programming languages. In particular, denotations do not usually incorporate program phrases as components. In fact it would not conflict with compositionality to let phrases denote even themselves, but such "denotations" tend to have (extremely) poor abstractness.

There are two cases, however, when it *is* desirable to use phrases as denotations:

- *identifiers* usually have to be their own denotations (e.g., in declarations); and
- for languages like LISP, where *phrases* can be *computed*, the denotation of a phrase essentially corresponds to its abstract syntax (and the benefits of the denotational approach are then questionable, since semantic equivalence is just syntactic identity).

2.3.2 Semantic Functions

Semantic functions map phrases (of abstract syntax) to their actual denotations. The semantics of a programming language may be specified by defining a semantic function for each sort of phrase.

Recall that abstract syntactic entities have an unambiguous structure. Hence semantic functions may be defined *inductively* by specifying, for each syntactic construct, its denotation in terms of the denotations of its components (if there are any). The conventional way of writing such an inductive definition in Denotational Semantics is as a set of so-called *semantic equations*, with (in general) one semantic equation for each production of the abstract syntax.

Let a, a_1, ..., a_n be (possibly-subscripted) nonterminal symbols, with associated phrase sorts s, s_1, ..., s_n. Let $\mathcal{F}_s$, $\mathcal{F}_{s_1}$, ..., $\mathcal{F}_{s_n}$ be semantic functions mapping phrases of sort $s_{(i)}$ to their

denotations (in practice, the semantic functions are usually given mnemonic names when they are introduced). Then the semantic equation for the production '$a ::= u_0 a_1 \ldots a_n u_n$' is of the form

$$\mathcal{F}_s[\![\ u_0 a_1 \ldots a_n u_n\ ]\!] \;=\; f(\mathcal{F}_{s_1}[\![a_1]\!], \ldots, \mathcal{F}_{s_n}[\![a_n]\!]).$$

The way that the denotations of the phrases $a_1, \ldots, a_n$ are combined is expressed using whatever notation is available for specifying particular objects—determining a function, written f above.

Note that the emphatic brackets [ ] separate the realm of syntax from that of semantics, which avoids confusion when programming languages contain the same mathematical notations as are used for expressing denotations.

To illustrate the form of semantic equations, let us specify denotations for signed binary numerals (with the abstract syntax given in Table 2.5). We take for granted the ordinary mathematical notation $(0, 1, 2, +, -, \times)$ for specifying particular integers in Z and natural numbers in N. The semantic functions ($\mathcal{Z}$ for signed binary numerals, $\mathcal{B}$ for unsigned binary numerals) are defined inductively by the semantic equations given in Table 2.6.

$\mathcal{Z}$: SIGNED-BINARY-NUMERAL $\rightarrow$ Z

$$\mathcal{Z}[\![\ B\ ]\!] \;=\; \mathcal{B}[\![B]\!]$$

$$\mathcal{Z}[\![\ \texttt{-}\ B\ ]\!] \;=\; -\mathcal{B}[\![B]\!]$$

$\mathcal{B}$: BINARY-NUMERAL $\rightarrow$ N

$$\mathcal{B}[\![\ \texttt{0}\ ]\!] \;=\; 0$$

$$\mathcal{B}[\![\ \texttt{1}\ ]\!] \;=\; 1$$

$$\mathcal{B}[\![\ B\ \texttt{0}\ ]\!] \;=\; 2 \times (\mathcal{B}[\![B]\!])$$

$$\mathcal{B}[\![\ B\ \texttt{1}\ ]\!] \;=\; (2 \times (\mathcal{B}[\![B]\!])) + 1$$

Table 2.6: Denotations for signed binary numerals

Perhaps the standard interpretation of binary notation is so much taken for granted that we may seem to be merely "stating the obvious" in the semantic equations. But we could just as well have specified alternative interpretations, e.g., by reversing the rôles of '0' and '1', or by making the right-most bit the most significant.

In effect, the semantic equations *reduce* the semantics of the language described (here, the binary numerals) to that of a "known" language (here, that of ordinary arithmetic). This reduction may also be viewed as a "syntax-directed translation", although it is then essential to bear in mind that phrases are semantically-equivalent whenever they are translated to notation that has the same *interpretation*, not merely the same *form*.

An alternative way of specifying semantic functions is to exploit the formulation of abstract syntax as an initial algebra, discussed in Section 2.2. Recall that the abstract syntax $\mathcal{A}(G)$ specified by a grammar G is a Σ_G-algebra, where Σ_G is the signature corresponding to the productions of G. As $\mathcal{A}(G)$ is the initial Σ_G-algebra (Proposition 32), there is a unique Σ_G-homomorphism from $\mathcal{A}(G)$ to any other Σ_G-algebra. So all that is needed is to make the spaces of denotations into a "target" Σ_G-algebra, say D, by defining a function p_G for each $p \in \Sigma_G$, i.e., for each production p of G. Then the semantic functions are given as the components of the unique Σ_G-homomorphism from $\mathcal{A}(G)$ to D.

This approach is known as *Initial Algebra Semantics*. Whereas such an explicit algebraic formulation can be convenient for some purposes, the approach is essentially the same as Denotational Semantics, and it is a simple matter to transform semantic equations into specifications of target algebras—or *vice versa*—while preserving the semantic functions that are thereby defined.

2.3.3 Notational Conventions

Some abbreviatory techniques are commonly used in semantic equations:

- The semantics of a construct may be specified in terms of that of a compound phrase, provided no circularity is introduced into the inductive definition. For instance, we might specify

  $$\mathcal{S}[\![\ \texttt{if}\ E\ \texttt{then}\ S\ ]\!] = \mathcal{S}[\![\ \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else skip}\ ]\!]$$

  where $\mathcal{S}[\![\ \texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ ]\!]$ is specified by an ordinary semantic equation. As well as abbreviating the right-hand sides of semantic equations, the use of this technique emphasizes that the syntactic construct is just "syntactic sugar" and does not add anything of (semantic) interest to the language.

- There may be several semantic functions for a single phrase sort, say $\mathcal{F}^i : P \to D_i$. This corresponds to a single function $\mathcal{F} : P \to (D_1 \times \ldots \times D_n)$, with the components of denotations being defined separately.

- The names of semantic functions may be omitted (when there is no possibility of confusion). In particular, when identifiers are essentially their own denotations, their (injective) semantic function is generally omitted.

These abbreviations have been found to increase the readability of denotational descriptions without jeopardizing their formality.

2.4 Domains

Appropriate mathematical spaces for the denotations of programming constructs are called *(semantic) domains*. Here, after a brief introduction to the basic concept of a domain, a summary is given of the notation used for specifying domains and their elements. A thorough explanation of the notation, together with the *theory* of domains, is given by Gunter and Scott [18]. The main *techniques* for choosing domains for use in denotational descriptions of programming languages are demonstrated in Section 2.5.

2.4.1 Domain Structure

Domains are sets whose elements are partially-ordered according to their degree of "definedness". When x is less defined than y in some domain D, we write $x \sqsubseteq_D y$ and say that x *approximates* y in D. (Mention of the domain concerned may be omitted when it is clear from the context.) Every domain D is assumed to have a *least element* $\bot_D$, representing "undefinedness"; moreover there are *limits* $\bigsqcup_n x_n$ for all (countable) increasing sequences $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$. (Thus domains are so-called *(ω-)cpos*. Further conditions on domains are imposed by Gunter and Scott [18]; but these conditions need not concern us here, as their primary purpose is to ensure that the class of domains is closed under various constructions.)

For an example, consider the set of *partial functions* from N to N, and for partial functions f, g, let $f \sqsubseteq g$ iff $\mathbf{graph}(f) \subseteq \mathbf{graph}(g)$. This gives a domain: $\sqsubseteq$ is a partial order corresponding to definedness; the least element $\bot$ is the empty function (but every total function is maximal, so there is no greatest element); and the limit of any increasing sequence of functions is given by the union of the graphs of the functions.

A domain D may be defined simply by specifying its elements and it approximation relation $\sqsubseteq$, as above. But it is tedious to check each time that $\sqsubseteq$ has the required properties—and to define *ad hoc* notation for identifying elements.

In practice, the domains used in denotational descriptions are generally defined as solutions (up to isomorphism) of *domain equations* involving the standard primitive domains and standard domain constructions. Not only does this ensure that the defined structures really are domains, it also provides us with standard notation for their elements.

The standard *primitive domains* are obtained merely by adding $\bot$ to an unordered (but at most countable) set, of course letting $\bot \sqsubseteq x$ for all x. Domains with such a trivial structure are called *flat*. For example the domain T of truth values is obtained by adding $\bot$ to the set $\{\mathsf{true}, \mathsf{false}\}$, and the domain $\mathsf{N}_\bot$ of natural numbers by adding $\bot$ to N.

There are standard domain constructions that correspond closely to well-known set constructions: Cartesian product, disjoint union, function space, and power sets. Of course these domain constructions have to take account of the $\sqsubseteq$ relation, as well as the elements; this leads to several possibilities.

Before proceeding to the details of the standard domain notation, let us consider what *functions* between domains are required.

The functions generally needed for the semantics of programming languages are *monotone*, in that they preserve the relation $\sqsubseteq$:

$x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$;

and *continuous*, in that they also preserve limits of increasing sequences:

$x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ implies $f(\bigsqcup_n x_n) = \bigsqcup_n f(x_n)$.

Note that we do *not* insist that functions preserve least elements. Those functions f that do satisfy

$f(\bot) = \bot$

are called *strict*. Constant functions are non-strict functions (in general).

Partial functions from N to N are represented by strict total functions from $\mathsf{N}_\bot$ to $\mathsf{N}_\bot$, the result $\bot$ corresponding to "undefined". Notice, by the way, that *all* strict functions on $\mathsf{N}_\bot$ are continuous—but in practice, we only make use of those that are computable in the usual sense.

The importance of continuity is two-fold:

(i) Let D be a domain. For any continuous function $f : D \to D$ there is a least $x \in D$ such that

$x = f(x)$.

This x is called the *least fixed point* of f, written $\mathsf{fix}(f)$. It is given by $\bigsqcup_n f^n(\bot)$.

Non-monotone functions on domains need not have fixed points at all; whether monotone (but non-continuous) functions on domains always have least fixed points depends on the precise structure of domains, but in any case their fixed points are not necessarily obtainable as the limits of countable increasing sequences.

(ii) There exists a non-trivial domain D_∞ such that

$\mathsf{D}_\infty \cong \mathsf{D}_\infty \to \mathsf{D}_\infty$

provided that $\mathsf{D}_\infty \to \mathsf{D}_\infty$ is just the space of *continuous* functions on D_∞. Domains such as D_∞ that "contain" their own (continuous) function space are called *reflexive*.

If $\mathsf{D}_\infty \to \mathsf{D}_\infty$ were to be the space of *all* functions, D_∞ would *have* to be the trivial (one-point) domain, by Cantor's Theorem.

Least fixed points of continuous functions provide appropriate denotations for iterative and recursive programming constructs. Reflexive domains are needed for the denotations of constructs that may involve "self-application": procedures with procedure parameters in ALGOL60, functions with dynamic bindings in LISP, assignments of procedures to variables in C, etc. Even when self-application is forbidden (e.g., by type constraints), it may still be simpler to specify denotations as elements of reflexive domains, rather than to introduce infinite families of non-reflexive domains.

The structure of domains described above is further motivated by the fact that domains with continuous functions provide denotations for almost all useful programming constructs. (The only exception seems to be constructs that involve so-called "unbounded nondeterminism", corresponding to infinite sets of implementation-dependent choices.)

2.4.2 Domain Notation

Now for a summary of the notation for specifying domains and their elements, following Gunter and Scott [18]. The (abstract) syntax of the notation is given in Table 2.7. Some conventions for disambiguating the written representation of the notation, together with some abbreviations for commonly-occurring patterns of notation, are given in Section 2.4.3.

(DOMAIN-EXPRESSIONS)

$$\begin{aligned} d \;::=\;& w \mid \mathsf{I} \mid \mathsf{O} \mid \mathsf{T} \mid \mathsf{N}_\perp \mid \\ & d_1 \to d_2 \mid d_1 \multimap d_2 \mid \\ & d_1 \times \cdots \times d_n \mid d_1 \otimes \cdots \otimes d_n \mid \\ & d_1 + \cdots + d_n \mid d_1 \oplus \cdots \oplus d_n \mid \\ & d_\perp \mid d^* \mid d^\infty \mid d^\natural \end{aligned}$$

(DOMAIN-VARIABLES)

$$w \;::=\; \textit{arbitrary symbols}$$

(EXPRESSIONS)

$$\begin{aligned} e \;::=\;& x \mid \perp_d \mid \top \mid \mathsf{true} \mid \mathsf{false} \mid \\ & e_1 =_d e_2 \mid \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 \mid 0 \mid \mathsf{succ} \mid \\ & \lambda x \in d.\, e \mid e_1\, e_2 \mid \mathsf{id}_d \mid e_1 \circ e_2 \mid \mathsf{fix}_d \mid \mathsf{strict}_d \mid \\ & (e_1, \ldots, e_n) \mid \langle e_1, \ldots, e_n \rangle \mid \mathsf{on}_i^d \mid \mathsf{smash}_d \mid \\ & [e_1, \ldots, e_n] \mid \mathsf{in}_i^d \mid \mathsf{up}_d \mid \mathsf{down}_d \mid \\ & \{\!|e|\!\} \mid e_1 \uplus e_2 \mid e^\natural \mid \mathsf{ext}_d \end{aligned}$$

(VARIABLES)

$$x \;::=\; \textit{arbitrary symbols}$$

Table 2.7: Notation for domains and elements

Let us start with *domain expressions*, d. These may include references to domain *variables*, w, whose interpretation is supplied by the context of the domain expression. This context is generally a set of *domain equations* of the form

$$w_1 = d_1, \ldots, w_n = d_n$$

where the w_i are distinct and no other variables occur in the d_i. As is shown by Gunter and Scott [18], there is always a "minimal" solution to such a set of equations (up to isomorphism). We need not worry here about the construction of the solution—the equations themselves express all that we really need to know about the defined domains. Note, however, that the simple equation

$$D = D \to D$$

defines D to be the trivial (one-point) domain! Most domain equations that arise in practice do not admit trivial solutions. (Gunter and Scott [18] show how to force non-trivial solutions to $D = D \to D$.)

Element expressions e may include references to element variables x whose domain and interpretation is supplied by the context. Usually the context is just the enclosing (element) expression, but we also allow *auxiliary* definitions of the form

$$x = e \in d$$

The scope of an auxiliary definition is the entire specification. (Mutually-dependent auxiliary definitions may be regarded as abbreviations for independent definitions involving the least fixed point operator.)

Basic Domains

$\bot_d$ denotes the least element of a domain d.

I is the 1-point domain, consisting only of $\bot_{\mathsf{I}}$.

O is the 2-point domain, consisting of $\bot_{\mathsf{O}}$ and $\top$. (Domains do not usually have greatest elements, so there is no need for a general notation $\top_d$.)

Truth Values

T is the flat 3-point domain of truth values, consisting of $\bot_{\mathsf{T}}$, **true**, and **false**.

$e_1 =_d e_2$ tests the equality of e_1 and e_2 in any *flat* domain d. The value is $\bot_{\mathsf{T}}$ if either or both of e_1 and e_2 denote $\bot_d$; otherwise it is **true** or **false**. (The monotonicity and continuity of $=_d$ follow from the flatness of d: equality would not be monotonic on a non-flat domain.)

if e_1 **then** e_2 **else** e_3 requires e_1 to denote an element of T, and e_2, e_3 to denote elements of some domain d. Then it denotes e_2 if e_1 denotes **true**; it denotes e_3 if e_1 denotes **false**; and it denotes $\bot_d$ if e_1 denotes $\bot_{\mathsf{T}}$.

In practice we allow all the usual Boolean functions, extended strictly (in all arguments) to T, and written using infix notation.

Natural Numbers

$\mathsf{N}_\bot$ is the flat domain of natural numbers, consisting of $\bot_{\mathsf{N}_\bot}$, 0, 1, ... (no infinity).

succ denotes the strict extension of the successor function from N to $\mathsf{N}_\bot$.

In practice we allow all known (computable) functions on the natural numbers, extended strictly in all arguments to $\mathsf{N}_\bot$, and written using infix notation.

Function Domains

$d_1 \to d_2$ denotes the domain of all continuous functions from the domain denoted by d_1 to the domain denoted by d_2. (Henceforth the tedious "denoted by" is generally omitted.) We have $f \sqsubseteq_{d_1 \to d_2} g$ iff $f(x) \sqsubseteq_{d_2} g(x)$ for all x in d_1.

$\lambda x \in d.\ e$ denotes the (continuous) function f given by defining $f(x) = e$, where x ranges over d. This provides the context for interpreting references to x in e.

$e_1\, e_2$ denotes the result $f(x)$ of applying the function $f : d \to d'$ denoted by e_1 to the value $x \in d$ denoted by e_2.

id_d denotes the identity function on domain d.

$e_1 \circ e_2$ denotes the composition of the functions $f_1 : d' \to d''$ and $f_2 : d \to d'$ denoted by e_1 and e_2, respectively, so that for all $x \in d$, $(e_1 \circ e_2)(x) = e_1(e_2(x))$.

fix_d denotes the least fixed point operator for domain d, which maps each function f in $d \to d$ to the least solution x of the equation $x = f(x)$.

$d_1 \circ\!\!\to d_2$ denotes the restriction of $d_1 \to d_2$ to strict functions.

strict_d denotes the function that maps each function in $d_1 \to d_2$ to the corresponding strict function in d, where $d = d_1 \circ\!\!\to d_2$.

Product Domains

$d_1 \times \cdots \times d_n$ denotes the Cartesian product domain of n-tuples, for any $n \geq 2$, generalizing the binary product domain of pairs. We have $(x_1, \ldots, x_n) \sqsubseteq_{d_1 \times \cdots \times d_n} (y_1, \ldots, y_n)$ iff $x_i \sqsubseteq_{d_i} y_i$ for $i = 1, \ldots, n$.

$(e_1, \ldots, e_n)$ denotes the n-tuple with components $e_1, \ldots, e_n$, for any $n \geq 2$.

$\langle e_1, \ldots, e_n \rangle$ denotes the target tupling of the functions denoted by the e_i, abbreviating $\lambda x \in d.\ (e_1(x), \ldots, e_n(x))$, where x does not occur in the e_i.

on_i^d denotes the projection onto the i'th component, mapping $(x_1, \ldots, x_n)$ to x_i , where $d = d_1 \times \cdots \times d_n$.

$d_1 \otimes \cdots \otimes d_n$ denotes the "smash" product obtained from the Cartesian product by identifying all the n-tuples that have any $\perp$ components. Note that $\otimes$ preserves flatness of domains.

smash_d denotes the function that maps each element of d, where $d = d_1 \times \cdots \times d_n$, to the corresponding element of $d_1 \otimes \cdots \otimes d_n$, giving $\perp$ if any of the components are $\perp$.

Sum Domains

$d_1 + \cdots + d_n$ denotes the "separated" sum domain d whose elements are (distinguished copies of) the elements of the d_i together with a new $\bot_d$. Elements of d originating from different summands d_i are incomparable in d.

$d_1 \oplus \cdots \oplus d_n$ denotes the "coalesced" sum domain d where the $\bot$ elements of (the distinguished copies of) the d_i are identified with $\bot_d$. Note that $\oplus$ preserves flatness of domains.

in_i^d denotes the injection function mapping elements of d_i to the corresponding elements of d, where $d = d_1 \oplus \cdots \oplus d_n$ and $1 \leq i \leq n$.

$[e_1, \ldots, e_n]$ denotes the "case analysis" of the functions $f_i : d_i \to d'$ denoted by the e_i, mapping $\mathsf{in}_i(x)$ to the value of $f_i(x)$ for $1 \leq i \leq n$ (but mapping $\bot$ to $\bot$).

Lifted Domains

$d_\bot$ denotes the lifted domain d' obtained by adding a new $\bot_{d'}$ under (a distinguished copy of) d.

up_d denotes the function that maps each element of d to the corresponding element of $d_\bot$.

down_d denotes the function that maps each element of $d_\bot$ back to the corresponding element of d.

Lists

d^* denotes the domain of lists of *finite* length, with non-$\bot$components in d. Note that lists with different lengths are incomparable in $\sqsubseteq$.

d^∞ denotes the domain of *infinite* lists with components in d. Here, the "empty" list is the infinite list of $\bot$'s. We let $l_1 \sqsubseteq_{d^\infty} l_2$ iff every component of l_1 approximates the corresponding component of l_2. Thus the empty list approximates all other lists.

Power Domains

$d^\natural$ denotes the "natural" (convex, Plotkin) power domain. Its elements may be imagined as *equivalence classes* of sets, where two sets are equivalent iff this follows from the continuity (and associativity, commutativity and absorption) of the binary union operation.

E.g., if $x \sqsubseteq y \sqsubseteq z$, then the sets $\{x, y, z\}$ and $\{x, z\}$ are equivalent; moreover, if $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ and $X = \{x_n \mid 0 \leq n\}$, then X is equivalent to $X \cup \{\bigsqcup_n x_n\}$.

The other power domains (upper, lower) considered by Gunter and Scott [18] are not used in this chapter: they do not accurately reflect the *possibility* of non-termination, as they force sets $X \cup \{\bot\}$ to be equivalent either to $\{\bot\}$ or to X.

$\{\!|e|\!\}$ denotes the element of $d^\natural$ corresponding to the set $\{x\}$, where e denotes the element $x \in d$.

$e_1 \uplus e_2$ denotes the element of $d^\natural$ corresponding to the union of X_1 and X_2, where e_1 and e_2 denote elements of a power domain $d^\natural$ corresponding to the sets X_1 and X_2.

$e^\natural$ denotes the pointwise extension of e to map $d_1{}^\natural \to d_2{}^\natural$, where e denotes a function in $d_1 \to d_2$.

$\mathbf{ext}_d$ extends functions in $d = d_1 \to d_2{}^\natural$ to $d_1{}^\natural \to d_2{}^\natural$.

It is possible to represent the *empty set* by using the domain $\mathsf{O} \oplus d^\natural$ instead of just $d^\natural$; emptiness can be tested for using $[\lambda x \in \mathsf{O}.\ e_1, \lambda x \in d^\natural.\ e_2]$. However, there is *no* (continuous) test for *membership* in power domains (just as there is no continuous test for equality on non-flat domains).

So much for the basic notation for domains and their elements.

2.4.3 Notational Conventions

When the above notation is written in semantic descriptions, domain expressions in element expressions are generally omitted when they can be deduced from the context. Parentheses are used to indicate grouping, although the following conventions allow some parentheses to be omitted:

- Function domain constructions $\to$ and $\circ\!\!\to$ associate to the right, and have weaker precedence than $+$, $\oplus$, $\times$, and $\otimes$:
  $D_1 \times D_2 \to D_3 \to D_4$ is grouped as $(D_1 \times D_2) \to (D_3 \to D_4)$.
- Application is left-associative, and has higher precedence than the other operators: $f\,x\,y$ is grouped as $(f\,x)\,y$, and $f \circ g(x)$ is grouped as $f \circ (g(x))$;
- Abstraction $\lambda x \in d.\ e$ extends as far as possible: $(\lambda x \in D.\ f\,x)$ is grouped as $(\lambda x \in D.\ (f\,x))$;
- Composition $\circ$ is associative, so its iteration does not need grouping.

(Without these conventions, our semantic descriptions would require an uncomfortable number of parentheses.) Furthermore, when implied unambiguously by the context, the following operations may be omitted:

- isomorphism between w and d, when $w = d$ is a specified domain equation;
- the following isomorphisms (which follow from the definitions of the basic domains and domain constructors):
  - $d_1 + \cdots + d_n \cong (d_1)_\perp \oplus \cdots \oplus (d_n)_\perp$;
  - $\mathsf{O} \cong \mathsf{I}_\perp$;
  - $\mathsf{T} \cong (\mathsf{O} \oplus \mathsf{O})$, mapping **true** to $\mathsf{in}_1(\perp_\mathsf{O})$;
  - $\mathsf{N}_\perp \cong (\mathsf{O} \oplus \mathsf{N}_\perp)$, mapping 0 to $\mathsf{in}_1(\perp_\mathsf{O})$;
  - $d^* \cong (\mathsf{O} \oplus (d \otimes d^*))$;
  - $d^\infty \cong (d \times d^\infty)$;

– $d^\infty \cong (\mathsf{N}_\perp \multimap d)$;

- injections $\mathsf{in}_i : d_i \multimap d_1 \oplus \cdots \oplus d_n$;
- "bottom extensions" of functions $f : d_i \to d'$ to sum domains: $[\ldots, \perp, f, \perp, \ldots] : d_1 \oplus \cdots \oplus d_n \multimap d'$;
- the inclusions of $d_1 \otimes \cdots \otimes d_n$ in $d_1 \times \cdots \times d_n$ and of $d \multimap d'$ in $d \to d'$, and the strict inclusion of $d_1 \oplus \cdots \oplus d_n$ in $d_1 + \cdots + d_n$.

Finally, the notation $\lambda(x_1 \in d_1, \ldots, x_n \in d_n).\ e$ abbreviates

$$\lambda x \in d_1 \times \cdots \times d_n.\ (\lambda x_1 \in d_1. \cdots \lambda x_n \in d_n.\ e)(\mathsf{on}_1 x) \cdots (\mathsf{on}_n x)$$

(where x does not occur in e). It denotes the function f defined by $f(x_1, \ldots, x_n) = e$.

2.5 Techniques

The preceding sections introduced all the *formalism* that is needed for specifying denotational descriptions of programming languages: grammars, for specifying abstract syntax; domain notation, for specifying domains and their elements; and semantic equations, for specifying semantic functions mapping syntactic entities to their denotations.

This section gives some examples of denotational descriptions. The main purpose of the examples is to show what *techniques* are available for modeling the fundamental concepts of programming languages (sequential computation, scope rules, local variables, etc.).

Familiarity with these techniques allows the task of specifying a denotational semantics of a language to be factorized into (i) analyzing the language in terms of the fundamental concepts, and (ii) combining the techniques for modeling the concepts involved. Furthermore, the understanding of a given denotational description may be facilitated by recognition of the use of the various techniques.

The programming constructs dealt with in the examples below are, in general, *simplified* versions of constructs to be found in conventional "high-level" programming languages. It is not claimed that the agglomeration of the exemplified constructs would make a particularly elegant and/or practical programming language.

Section 2.5.1 outlines the semantics of *literals* (numerals, strings, etc.). Then Section 2.5.2 specifies denotations for arithmetical and Boolean *expressions*, illustrating a simple technique for dealing with "errors". Section 2.5.3 shows how to specify denotations for *constant declarations*, using "environments" to model scopes. Section 2.5.4 extends expressions to include *function abstractions*, and gives a denotational description of the *λ-calculus*.

Next, Section 2.5.5 gives denotations for *variable declarations*, using "stores" and "locations". Then Section 2.5.6 deals with *statements*, using "direct" semantics; it also explains how the technique of "continuations" can be used to model *jumps*. Section 2.5.7 describes *procedures* with various *modes* of parameter evaluation.

Section 2.5.8 distinguishes between the concepts of "batch" and "interactive" *input and output*. Section 2.5.9 shows how powerdomains can be used to model *nondeterministic programs*. Finally, Section 2.5.10 introduces "resumptions" and uses them to give denotations for a simple form of *concurrent processes*.

Caveat: In Section 2.5.2, the denotations of expressions are simply (numerical, etc.) values. But later, they have to be *changed*: in Section 2.5.3 (to be functions of environments), and again in Section 2.5.5 (to be functions of stores). Such changes to denotations entail tedious changes to the semantic equations that involve them. This rather unfortunate feature of conventional denotational descriptions stems from the fact that the notation used in the semantic equations has to match the precise domain structure of denotations.

Of course, these changes would be unnecessary if denotations of expressions were to be functions of environments and stores from the start. Although that might be appropriate when giving a denotational description of a complete programming language, it is undesirable in this introduction:

the complexity of the denotations of simple constructs would obscure the relation between particular program constructs and the *appropriate* techniques for modeling them.

An alternative approach is to introduce *auxiliary notation* for *combining denotations*. Then when domains of denotations are changed, only the definition of the auxiliary notation requires modification: the semantic equations themselves may be left unchanged. Moreover, the auxiliary notation may be chosen to correspond directly to fundamental concepts, such as "sequencing" and "block structure", so that the semantic equations explicate the fundamental conceptual analysis of the described constructs. Such an approach is presented elsewhere [34]. It would be inappropriate to adopt it here, as it tends to hide the mathematical essence of denotations, and would give a distorted impression of the conventional approach to Denotational Semantics.

2.5.1 Literals

The syntax of a programming language usually includes "literals" (sometimes called "literal constants", or just "constants"). A literal is a symbol (or phrase) that always refers to the same item of data, irrespective of where it occurs. Examples of literals are '`true`' and '`false`', numerals, characters, and character strings.

The denotational semantics of literals is fairly straightforward, but somewhat tedious, to specify. We have already seen a simple example: binary numerals (Section 2.3). So let us skip most of the details here. A skeleton abstract syntax for literals is given in Table 2.8.

(LITERAL)
L ::= `true` | `false` | N | C | CS
(NUMERAL)
N ::= *unspecified*
(CHARACTER)
C ::= *unspecified*
(CHARACTER-STRING)
CS ::= *unspecified*

Table 2.8: Syntax for literals

For the denotations of '`true`' and '`false`', we may use the values **true** and **false** of the standard domain T. The denotations of numerals should take into account that different implementations generally impose different bounds on the magnitude of numbers, and on the accuracy of "real" numbers. So let the domain of numbers—together with the associated operations—be a *parameter* of the semantics. The same goes for the denotations of characters (the ordering may vary between implementations) and strings (their length may be bounded). Let us leave such parameters as unspecified variables in the semantic description.

For example, let the domains **Num**, **Char**, and **String** be unspecified domain variables, together

with various variables for elements of, and functions on, these domains; see Table 2.9. It is straightforward to define the semantic functions introduced in Table 2.10 in terms of the given elements and functions. The details are omitted here.

Num	=	*unspecified*
zero, one	$\in$	Num
neg	$\in$	Num $\circ\!\!\rightarrow$ Num
sum, diff	$\in$	(Num $\otimes$ Num) $\circ\!\!\rightarrow$ Num
prod, div	$\in$	(Num $\otimes$ Num) $\circ\!\!\rightarrow$ Num
Char	=	*unspecified*
ord	$\in$	Char $\circ\!\!\rightarrow$ Num
chr	$\in$	Num $\circ\!\!\rightarrow$ Char
String	=	*unspecified*
str	$\in$	Char* $\circ\!\!\rightarrow$ String
chrs	$\in$	String $\circ\!\!\rightarrow$ Char*
V	=	T $\oplus$ Num $\oplus$ Char $\oplus$ String

Table 2.9: Domains for literals

$\mathcal{L}$: LITERAL $\rightarrow$ V
$\mathcal{N}$: NUMERAL $\rightarrow$ Num
$\mathcal{C}$: CHARACTER $\rightarrow$ Char
$\mathcal{CS}$: CHARACTER-STRING $\rightarrow$ String

Table 2.10: Denotations for literals

By the way, the domains of literal denotations are generally *flat* (and countable). Note in particular that the finite *numerical* approximations to real numbers made by computers should *not* be represented by values related by the *computational* approximation ordering of domains, $\sqsubseteq$:

once an approximate real number has been computed, further computation does not improve the degree of approximation of *that* number. (Of course, a program may indeed compute a series of approximate numbers, but the numbers are not *necessarily* increasingly-good approximations to some particular number.)

2.5.2 Expressions

Expressions in programming languages are constructed using operators and (perhaps) if-then-else from primitive expressions, including literals. Abstract syntax for some typical expressions is given in Table 2.11. (Further expressions are considered in later sections.)

(EXPRESSION)
E ::= L | $MO\ E_1$ | $E_1\ DO\ E_2$ | `if` E_1 `then` E_2 `else` E_3

(MONADIC-OPERATOR)
MO ::= $\neg$ | $-$

(DYADIC-OPERATOR)
DO ::= $\wedge$ | $\vee$ | $+$ | $-$ | $*$ | $=$

Table 2.11: Syntax for expressions

We take the denotations of expressions to be elements of a domain EV that consists of truth-values, numbers, etc., representing the result of expression evaluation. The domain EV is a so-called "characteristic domain", and its relation to other characteristic domains introduced in later sections can give valuable insight into the essence of the described programming language. For now we let EV contain the same values as V, i.e., the values of literals; later, further expressible values are introduced.

We are now ready to define the denotations of expressions and operators; see Table 2.12. Note that the notational conventions introduced at the end of Section 2.4 are much exploited in the semantic equations. For instance, in the equation for if-expressions, there is an application of a function ($\lambda t \in \mathsf{T}. \ldots$) to an argument in EV; however, T is a summand of V, which is isomorphic to EV, so the given function, f say, is implicitly extended to $[f, \perp_{\mathsf{Num}\to\mathsf{EV}}, \perp_{\mathsf{Char}\to\mathsf{EV}}, \perp_{\mathsf{String}\to\mathsf{EV}}] \in (\mathsf{T} \oplus \mathsf{Num} \oplus \mathsf{Char} \oplus \mathsf{String}) \to \mathsf{EV}$, and then composed with an isomorphism to give a function in $\mathsf{EV} \to \mathsf{EV}$.

Thus the denotation of an erroneous expression such as '`if 42 then...else...`' is $\perp$. The semantics of such erroneous expressions is actually irrelevant, provided that programs containing them are deemed illegal. More generally, however, it might be better to avoid representing errors by $\perp$, as the *essential* use of $\perp$ (in later sections) is to represent *non-termination*. To do this we would have to introduce special elements for representing errors into *all* domains, and the extra notation for specifying the treatment of errors would be an unwelcome burden in the semantic equations.

$$\mathsf{EV} = \mathsf{V}$$

$$\mathcal{E} : \textsc{Expression} \rightarrow \mathsf{EV}$$

$$\mathcal{E}[\![\ L\ ]\!] = \mathcal{L}[\![L]\!]$$

$$\mathcal{E}[\![\ MO\ E_1\ ]\!] = \mathcal{MO}[\![MO]\!](\mathcal{E}[\![E_1]\!])$$

$$\mathcal{E}[\![\ E_1\ DO\ E_2\ ]\!] = \mathcal{DO}[\![DO]\!](\mathsf{smash}(\mathcal{E}[\![E_1]\!], \mathcal{E}[\![E_2]\!]))$$

$$\mathcal{E}[\![\ \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\ ]\!] = (\lambda t \in \mathsf{T}.\ \mathsf{if}\ t\ \mathsf{then}\ \mathcal{E}[\![E_2]\!]\ \mathsf{else}\ \mathcal{E}[\![E_3]\!])\ (\mathcal{E}[\![E_1]\!])$$

$$\mathcal{MO} : \textsc{Monadic-Operator} \rightarrow (\mathsf{V} \multimap \mathsf{V})$$

$$\mathcal{MO}[\![\ \neg\ ]\!] = \lambda t \in \mathsf{T}.\ \mathsf{if}\ t\ \mathsf{then\ false\ else\ true}$$

$$\mathcal{MO}[\![\ -\ ]\!] = \lambda n \in \mathsf{Num}.\ \mathsf{diff}(\mathsf{zero}, n)$$

$$\mathcal{DO} : \textsc{Dyadic-Operator} \rightarrow (\mathsf{V} \otimes \mathsf{V} \multimap \mathsf{V})$$

$$\mathcal{DO}[\![\ \wedge\ ]\!] = \lambda(t_1 \in \mathsf{T}, t_2 \in \mathsf{T}).\ \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else\ false}$$

$$\ldots$$

$$\mathcal{DO}[\![\ +\ ]\!] = \lambda(n_1 \in \mathsf{Num}, n_2 \in \mathsf{Num}).\ \mathsf{sum}(n_1, n_2)$$

$$\ldots$$

$$\mathcal{DO}[\![\ =\ ]\!] = \lambda(v_1 \in \mathsf{V}, v_2 \in \mathsf{V}).\ (v_1 =_{\mathsf{V}} v_2)$$

Table 2.12: Denotations for expressions

2.5.3 Constant Declarations

Identifiers are symbols used as "tokens" for values. In programming languages, there are various constructs which introduce identifiers and "bind" them to values. It is conventional to refer to the value to which an identifier is bound as the value "denoted" by the identifier, but this terminology is a bit misleading: the *denotation* of an identifier is the identifier itself (or rather, an element of a semantic domain corresponding to the abstract syntax of identifiers).

Let us start with some simple "constant declarations", whose abstract syntax is given in Table 2.13. The intended effect of the declaration '`val` I `=` E' is to "bind" I to the value of E. The construct '`let` CD `in` E' determines the "scope" of such "bindings": the bindings made by CD are available throughout E—except where overridden by another binding for the same identifier, since '`let`'s may be nested, giving a "block structure" in expressions. In 'CD_1 `;` CD_2', the scope of the bindings introduced by CD_1 includes CD_2. The phrase '`rec` CD' extends the scope of the declarations in CD to CD itself, making them "mutually-recursive".

(CONSTANT-DECLARATIONS)

CD ::= `val` I `=` E | CD_1 `;` CD_2 | `rec` CD

(EXPRESSION)

E ::= I | `let` CD `in` E

Table 2.13: Syntax for constant declarations

In the semantics, we write DV for the domain that represents the values "denotable" by identifiers. DV is a characteristic domain, like EV. In real programming languages there are sometimes values that are expressible but not denotable—numbers in ALGOL60, for instance. Less obviously, there may be values that are denotable but not expressible—types in PASCAL, for instance.

"Environments" are used to represent associations between identifiers and denoted values. The domain of environments, together with some basic functions on environments, is defined in Table 2.14. Ide is assumed to be a flat domain corresponding to the abstract phrase sort IDENTIFIER. The element $\top \in \mathsf{O}$ is used to indicate the absence of a denoted value. (To allow the presence of a denoted value to be tested, we would have to lift DV to $\mathsf{DV}_\perp$, since the denoted value might be $\perp$.) Notice that $\mathsf{overlay}(e, e')$ gives precedence to e, whereas $\mathsf{combine}(e, e') = \mathsf{combine}(e', e)$ is intended for uniting the bindings of disjoint sets of identifiers.

The result of expression evaluation now depends, in general, on the values bound to the identifiers that occur in it. This dependence is represented by letting the denotation of an expression be a *function* from environments to expressible values—which requires rewriting the semantic equations previously specified for expressions.

Clearly, an appropriate denotation for a constant declaration is a function from environments to environments. But there is a choice to be made: should the resulting environment be the argument environment *extended* by the new bindings? or just the *new* bindings by themselves? Let us choose the latter, which gives a bit more flexibility, exploited in later sections.

$$
\begin{array}{lcl}
\mathsf{Env} & = & \mathsf{Ide} \to (\mathsf{DV} \oplus \mathsf{O}) \\
\mathsf{void} & = & \lambda I \in \mathsf{Ide}.\ \mathsf{in}_2\, \top \\
 & \in & \mathsf{Env} \\
\mathsf{bound} & = & \lambda I \in \mathsf{Ide}.\ \lambda e \in \mathsf{Env}.\ [\mathsf{id}_{\mathsf{DV}}, \bot](e(I)) \\
 & \in & \mathsf{Ide} \to \mathsf{Env} \to \mathsf{DV} \\
\mathsf{binding} & = & \lambda I \in \mathsf{Ide}.\ \lambda v \in \mathsf{DV}.\ \lambda I' \in \mathsf{Ide}.\ \mathsf{if}\ I =_{\mathsf{Ide}} I'\ \mathsf{then}\ \mathsf{in}_1(v)\ \mathsf{else}\ \mathsf{in}_2(\top) \\
 & \in & \mathsf{Ide} \to \mathsf{DV} \to \mathsf{Env} \\
\mathsf{overlay} & = & \lambda(e \in \mathsf{Env}, e' \in \mathsf{Env}).\ \lambda I \in \mathsf{Ide}.\ [\mathsf{id}_{\mathsf{DV}}, \lambda x \in \mathsf{O}.\ e'(I)](e(I)) \\
 & \in & \mathsf{Env} \times \mathsf{Env} \to \mathsf{Env} \\
\mathsf{combine} & = & \lambda(e \in \mathsf{Env}, \lambda e' \in \mathsf{Env}).\ \lambda I \in \mathsf{Ide}.\ [\lambda d \in \mathsf{DV}.\ [\bot, \lambda x \in \mathsf{O}.\ d], \lambda x \in \mathsf{O}.\ \mathsf{id}_{\mathsf{DV} \oplus \mathsf{O}}] \\
 & & \qquad\qquad (e(I))(e'(I)) \\
 & \in & \mathsf{Env} \times \mathsf{Env} \xrightarrow{\cdot} \mathsf{Env}
\end{array}
$$

Table 2.14: Notation for environments

The denotations of constant declarations and of the related expressions, together with the modified denotations of the previously-specified expressions, are defined in Table 2.15.

The semantics of recursive declarations makes use of $\mathsf{fix}_{\mathsf{Env}}$, which gives the least fixed point of the function in $\mathsf{Env} \to \mathsf{Env}$ to which it is applied. To see that this provides the appropriate denotations, consider $\mathcal{CD}[\![\ \texttt{rec val}\ I\ \texttt{=}\ E\ ]\!](e)$. From the semantic equations we have

$$
\begin{array}{l}
\mathcal{CD}[\![\ \texttt{rec val}\ I\ \texttt{=}\ E\ ]\!](e) = \\
\quad \mathsf{fix}(\lambda e' \in \mathsf{Env}.\ \mathsf{binding}(I)(\mathcal{E}[\![E]\!](\mathsf{overlay}(e', e))))
\end{array}
$$

i.e., the least $e' \in \mathsf{Env}$ such that

$$e' \;=\; \mathsf{binding}(I)(\mathcal{E}[\![E]\!](\mathsf{overlay}(e', e))).$$

Let $v = \mathcal{E}[\![E]\!](\mathsf{overlay}(e', e))$; we have

$$v \;=\; \mathcal{E}[\![E]\!](\mathsf{overlay}(\mathsf{binding}\ I\, v, e))$$

and in fact

$$v \;=\; \mathsf{fix}(\lambda v' \in \mathsf{EV}.\ \mathcal{E}[\![E]\!](\mathsf{overlay}(\mathsf{binding}\ I\, v', e))).$$

Notice that a direct circularity in the recursive declarations gives rise to $\bot$ as a denoted value, e.g.,

$$\mathsf{DV} = \mathsf{V}$$

$$\mathsf{EV} = \mathsf{V}$$

$$\mathcal{CD} : \textsc{Constant-Declarations} \to \mathsf{Env} \to \mathsf{Env}$$

$$\mathcal{CD}[\![\ \texttt{val}\ I\ \texttt{=}\ E\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{binding}\ I\ (\mathcal{E}[\![E]\!]e)$$

$$\mathcal{CD}[\![\ CD_1\texttt{;}\ CD_2\ ]\!] = \lambda e \in \mathsf{Env}.\ (\lambda e_1 \in \mathsf{Env}.\ \mathsf{overlay}(\mathcal{CD}[\![CD_2]\!](\mathsf{overlay}(e_1, e)), e_1))\ (\mathcal{CD}[\![CD_1]\!]e)$$

$$\mathcal{CD}[\![\ \texttt{rec}\ CD\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{fix}(\lambda e' \in \mathsf{Env}.\ \mathcal{CD}[\![CD]\!](\mathsf{overlay}(e', e)))$$

$$\mathcal{E} :\ \textsc{Expression} \to \mathsf{Env} \to \mathsf{EV}$$

$$\mathcal{E}[\![\ I\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{bound}\ I\, e$$

$$\mathcal{E}[\![\ \texttt{let}\ CD\ \texttt{in}\ E\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathcal{E}[\![E]\!](\mathsf{overlay}(\mathcal{CD}[\![CD]\!]e, e))$$

$$\mathcal{E}[\![\ L\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathcal{L}[\![L]\!]$$

$$\mathcal{E}[\![\ MO\ E_1\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathcal{MO}[\![MO]\!](\mathcal{E}[\![E_1]\!]e)$$

$$\mathcal{E}[\![\ E_1\ DO\ E_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathcal{DO}[\![DO]\!](\mathsf{smash}(\mathcal{E}[\![E_1]\!]e, \mathcal{E}[\![E_2]\!]e))$$

$$\mathcal{E}[\![\ \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\ ]\!] = \lambda e \in \mathsf{Env}.\ (\lambda t \in \mathsf{T}.\ \mathsf{if}\ t\ \mathsf{then}\ \mathcal{E}[\![E_2]\!]e\ \mathsf{else}\ \mathcal{E}[\![E_3]\!]e)\ (\mathcal{E}[\![E_1]\!]e)$$

Table 2.15: Denotations for constant declarations and expressions (modified)

$$\mathcal{CD}[\![\ \texttt{rec val}\ I\ \texttt{=}\ I\ ]\!] = \mathsf{binding}\ I\ \bot$$

in contrast to a mere "forward reference":

$$\mathcal{CD}[\![\ \texttt{rec (val}\ I\ \texttt{=}\ I'\texttt{; val}\ I'\ \texttt{=}\ \texttt{0)}\ ]\!] = \mathsf{overlay}(\mathsf{binding}\ I'\,0, \mathsf{binding}\ I\,0).$$

The most interesting case is when the sequence of environments e'_n defined by

$$\begin{aligned} e'_0 &= \mathsf{binding}(I)(\mathcal{E}[\![E]\!](\bot)) \\ e'_1 &= \mathsf{binding}(I)(\mathcal{E}[\![E]\!](\mathsf{overlay}(e'_0, e)) \\ &\cdots \\ e'_{n+1} &= \mathsf{binding}(I)(\mathcal{E}[\![E]\!](\mathsf{overlay}(e'_n, e)) \\ &\cdots \end{aligned}$$

is strictly increasing, converging to—but never reaching—the limit point $e' = \bigsqcup_n e'_n$. With the expressions considered so far, it is not possible to get such a sequence; but it becomes possible when *function abstractions* are introduced, as in the next section.

2.5.4 Function Abstractions

"Functions" in programs resemble mathematical functions: they return values when applied to arguments. In programs, however, the evaluation of arguments may diverge, so it is necessary to take into account not only the relation between argument values and result values, but also the stage at which an argument expression is evaluated: straight away, or when (if) ever the value of the argument is required for calculating the result of the application.

Various programming languages allow functions to be *declared*, i.e., bound to identifiers. Often, functions may also be passed as *arguments* to other functions. But only in a few languages is it possible to *express* functions directly, by means of so-called "abstractions", without necessarily binding them to identifiers. (These languages are generally the so-called "functional programming languages".)

The syntax given in Table 2.16 allows functions to be expressed by abstractions of the form '`fun (val` I`)` E'; we refer to '`val` I' as the "parameter declaration" of the abstraction (further forms of parameter declarations are introduced later) and to E as the "body". Notice that constant declarations of the form '`val` I' `= fun (val` I`)` E' resemble "function declarations" in conventional programming languages; recursive references to I' in E are allowed when the declaration is prefixed by '`rec`'.

The phrase '$E_1(E_2)$' expresses the application of a function to an argument, with the "actual parameter" E_2 being evaluated *before* the evaluation of the body of the function abstraction is commenced—this "mode" of parameter evaluation is known as "call by value". (Functions in programming languages are usually allowed to have lists of parameters; this feature is omitted here, for simplicity.)

(EXPRESSION)
E ::= `fun (`PD`)` E | $E_1(E_2)$
(PARAMETER-DECLARATION)
PD ::= `val` I

Table 2.16: Syntax for functions and parameter declarations

There are two distinct possibilities for the scopes of declarations in relation to abstractions, arising from identifiers which occur in the bodies of abstractions, but which refer to outer declarations. With so-called *static* scopes, the scopes of declarations extend into the bodies of an abstraction at the point where the abstraction is introduced, so that the declaration referred to by an identifier is fixed. With *dynamic* scopes, the body of an abstraction is evaluated in the scope of the declarations at each point of application, so that the declaration referred to by an identifier in an abstraction

body may vary—and be different from that referred to with static scopes. There is some dispute in the programming community about which of these scope rules is "better". Here, the semantic description of static scopes is illustrated; dynamic scopes are only marginally more complicated to describe.

The domains for use in the semantics of function abstractions are specified in Table 2.17. Notice that the definitions of DV and EV supercede the previous definitions. (No changes are needed to the semantic equations for declarations and expressions given in Table 2.12, thanks to our notational conventions about injections and extensions related to sums.)

$$\mathsf{F} = (\mathsf{PV} \multimap \mathsf{FV})_\perp$$

$$\mathsf{PV} = \mathsf{V} \oplus \mathsf{F}$$

$$\mathsf{FV} = \mathsf{V}$$

$$\mathsf{DV} = \mathsf{V} \oplus \mathsf{F}$$

$$\mathsf{EV} = \mathsf{V} \oplus \mathsf{F}$$

$$\mathcal{E} : \textsc{Expression} \to \mathsf{Env} \to \mathsf{EV}$$

$$\mathcal{E}[\![\ \texttt{fun}\ (PD)\ E\ ]\!] = \lambda e \in \mathsf{Env}.\ (\mathsf{up} \circ \mathsf{strict})(\lambda v \in \mathsf{PV}.\ \mathsf{id}_{\mathsf{FV}}(\mathcal{E}[\![E]\!](\mathsf{overlay}(\mathcal{PD}[\![PD]\!]v, e))))$$

$$\mathcal{E}[\![\ E_1(E_2)\ ]\!] = \lambda e \in \mathsf{Env}.\ (\mathsf{down} \circ \mathsf{id}_{\mathsf{F}})(\mathcal{E}[\![E_1]\!]e)(\mathcal{E}[\![E_2]\!]e)$$

$$\mathcal{PD} : \textsc{Parameter-Declarations} \to \mathsf{PV} \to \mathsf{Env}$$

$$\mathcal{PD}[\![\ \texttt{val}\ I\ ]\!] = \lambda v \in \mathsf{PV}.\ \mathsf{binding}\ I\ v$$

Table 2.17: Denotations for functions and parameter declarations

To model abstractions it is obvious to use functions. The domains consisting of parameter values, PV, and function result values, FV, may be regarded as characteristic domains. Few programming languages allow functions to be returned as results (and some even forbid functions as arguments).

The functions corresponding to the values of abstractions are taken to be *strict*, reflecting value-mode parameter evaluation: $\perp$ represents the non-termination of an evaluation, and the non-termination of an argument evaluation implies the non-termination of the function application. The abstraction values are *lifted* so that an abstraction never evaluates to $\perp$.

Notice that the domain F is *reflexive*: it is isomorphic to a domain that (essentially) includes a domain of functions from F.

The semantic equations for function abstractions are given in Table 2.17. Various isomorphisms

are left implicit, for instance that between DV and EV; likewise, some injections and extensions related to sum domains are omitted.

An alternative mode of parameter evaluation is to delay evaluation until the parameter is *used*. This mode is referred to as "call by name". (The main difference it makes to the semantics of expressions is that an evaluation which doesn't terminate with value-mode, *may* terminate when name-mode is used instead.)

Only a few programming languages provide name-mode parameters. Much the same effect, however, can be achieved by passing a (parameterless) abstraction as a parameter, and applying it (to no parameters) wherever the value of the parameter is required.

The main theoretical significance of name-mode abstractions is that they correspond directly to λ-abstractions in the *λ-calculus* of Church (see [4]). Consider the abstract syntax for λ-calculus expressions given in Table 2.18. The axiom of so-called "β-conversion" of the λ-calculus makes an application '$(\lambda I.\ E)(E')$' equivalent to the expression obtained by substituting E' for I in E (with due regard to static scopes of λ-bindings), and this is just E when I does not occur in E.

(EXPRESSION) E ::= $(\lambda I.\ E)$ | $E_1(E_2)$ | I (IDENTIFIER) I ::= *unspecified*

Table 2.18: Syntax for λ-expressions

It is a simple matter to adapt the domains that were used to represent value-mode abstractions, so as to provide a denotational semantics for the λ-calculus. The only necessary changes are to let FV include F, and to remove the restriction of F to strict functions; but let us dispense with the lifting as well, as it is no longer significant. The presence of V (in FV) ensures that the solution to the domain equations is non-trivial. (The *standard model* for the λ-calculus [18] is obtained by taking $\mathsf{PV} = \mathsf{FV} = \mathsf{F}$, leaving essentially $\mathsf{F} = \mathsf{F} \to \mathsf{F}$, and the trivial solution has to be avoided another way.)

The denotations for the λ-calculus are specified in Table 2.19, where for once the injections and extensions related to the sum domain are made explicit (although the isomorphisms between the left- and right-hand sides of the specified domain equations are still omitted).

The standard model for the λ-calculus has been extensively studied, and there are some significant theorems about it. Most of these carry over to the denotations defined above. First of all, there is the theorem that the semantics does indeed model β-conversion:

Proposition 33 *For any λ-expressions '$\lambda I.\ E$' and E',*

$$\mathcal{E}[\![\ (\lambda I.\ E)(E')\ ]\!] = \mathcal{E}[\![\ [E'/I]E\ ]\!].$$

Here '$[E'/I]E$' is the proper *substitution* of E' for free occurrences of I in E: the identifiers of λ-abstractions in E are assumed (or made) to be different from the free identifiers in E'.

$$\mathsf{F} = \mathsf{PV} \to \mathsf{FV}$$

$$\mathsf{PV} = \mathsf{V} \oplus \mathsf{F}$$

$$\mathsf{FV} = \mathsf{V} \oplus \mathsf{F}$$

$$\mathsf{DV} = \mathsf{V} \oplus \mathsf{F}$$

$$\mathsf{EV} = \mathsf{V} \oplus \mathsf{F}$$

$$\mathcal{E} : \textsc{Expression} \to \mathsf{Env} \to \mathsf{EV}$$

$$\mathcal{E}[\![\ (\lambda I.\ \ E)\ ]\!] =$$
$$\lambda e \in \mathsf{Env}.\ \mathsf{in}_2(\lambda v \in \mathsf{PV}.\ \mathcal{E}[\![E]\!](\mathsf{overlay}(\mathsf{binding}\ I\ v, e)))$$

$$\mathcal{E}[\![\ E_1(E_2)\ ]\!] = \lambda e \in \mathsf{Env}.\ [\bot, \mathsf{id}_\mathsf{F}](\mathcal{E}[\![E_1]\!]e)(\mathcal{E}[\![E_2]\!]e)$$

Table 2.19: Denotations for λ-expressions

The key to proving the above theorem is:

Lemma 34 (Substitution) *For any λ-expressions E, E', for any identifier I, and for any $e \in$ Env,*

$$\mathcal{E}[\![\ E\ ]\!](\mathsf{overlay}(\mathsf{binding}(I)(\mathcal{E}[\![E']\!]e), e)) \quad = \quad \mathcal{E}[\![\ [E'/I]E\ ]\!]e.$$

The following theorem implies that β-reduction is sufficient for symbolic computation of approximations to any desired degree of closeness. Let $\mathcal{A}(E)$ be the set of *approximate normal forms* of E (obtained from E by finite sequences of β-reductions, followed by the replacement of any remaining redexes by an expression 'Ω' denoting $\bot$).

Theorem 35 (Limiting Completeness) *For any λ-expression E,*

$$\mathcal{E}[\![E]\!] \quad = \quad \bigsqcup\{\mathcal{E}[\![E']\!] \mid E' \in \mathcal{A}(E)\}.$$

The original proof by Wadsworth [59] involves the introduction of an auxiliary calculus with numerical labels forcing all reduction sequences to terminate. An alternative proof is given by Mosses and Plotkin [36] by introducing an "intermediate" denotational semantics, where denotations are taken to be functions of an argument in the *chain* domain of extended natural numbers (i.e., with ∞): for finite arguments, the intermediate semantics gives approximations, corresponding to the denotations of approximate normal forms; the standard denotations are obtained when the argument is ∞.

2.5.5 Variable Declarations

The preceding sections dealt with expressions, constant declarations, and function abstractions. In conventional programming languages, these constructs play a minor rôle in comparison to *statements* (also called "commands"), which operate on "variables". This section deals with the semantics of variables; statements themselves are deferred to the next section.

In programs, variables are entities that provide access to stored data. The *assignment* of a value to a variable has the effect of modifying the stored data, whereas merely inspecting the current value of a variable causes no modification.

This concept of a variable is somewhat different from that of a variable in mathematics. In mathematical *terms*, variables stand for particular unknown values—often, the arguments of functions. These variables do indeed get "assigned" values, e.g., by function application. But the values thus assigned do *not* subsequently vary: a variable refers to the same value throughout the term in which it is used. In fact mathematical variables correspond closely to *identifiers* in programming languages.

Program variables may be *simple* or *compound*. The latter have component variables that may be assigned values individually; the value of a compound variable depends on the values of its component variables.

Consider the syntax specified in Table 2.20. The variable declaration '`var` I: T' determines a "fresh" variable for storing values of the "type" T, and binds I to the variable. Variable declarations are combined by 'VD_1 , VD_2'; such declarations do not include each other in their scopes (although in our simple example language, it would make no difference if they did, as variable declarations do not refer to identifiers at all). The types '`bool`', '`num`' are for declaring simple variables, for storing truth-values, respectively numbers; the type 'T`[1..`N`]`' is for declaring compound variables that have N independent component variables for storing values of type T. In the expression 'E_1`[`E_2`]`', E_1 is supposed to evaluate to a compound variable, v, and E_2 to a positive integer, n; then the result is the nth component variable of v.

(VARIABLE-DECLARATIONS)

VD ::= `var` I: T | VD_1 , VD_2

(TYPE)

T ::= `bool` | `num` | T`[1..`N`]`

(EXPRESSION)

E ::= E_1`[`E_2`]`

Table 2.20: Syntax for variable declarations and types

Types are used for two purposes in programming languages: to facilitate checking that programs are well-formed, prior to execution; and to indicate how much storage to allocate, during execution. Here, we are only concerned with the dynamic semantics of programs, which—in general—does not involve type checking, only storage allocation. (Mitchell [26] provides an extensive study of the

semantics of types.)

"Stores" are used to represent associations between *simple* variables and their values. Simple variables are represented by "locations" in stores; their only relevant property is that they can be distinguished from each other. Thus a simple variable identifier gets bound to a location, which in turn gives access to the current value stored in the variable. It is possible for two identifiers to be bound (in the same scope) to the same location: then assignment to the one changes the value of the other. Such identifiers are called "aliases".

Compound variables can be represented by values with variables (ultimately, locations) as components. Whereas assignment to distinct simple variables is independent, distinct compound variables may "share" component variables.

The domain of storable values, SV, consists of those items of data that can be stored at single locations. It may be considered to be a characteristic domain.

The domain of (states of) stores, S, is defined in Table 2.21, together with some basic functions on stores. A location mapped to **false** is "free", and a location mapped to **true** is "reserved" but not yet "initialized". Notice that the function '**location**' is left unspecified—it is supposed to select any location that is not reserved in the given state. It is usual to ignore the boundedness of real computer storage in denotational semantics, so '**location**' may be assumed not to produce $\perp$ (unless applied to a state in which all the locations have somehow been reserved).

Some further notation concerned with compound variables is specified in Table 2.22. It provides convenient generalizations of the basic functions on stores. LV is the domain of all variables; RV is the domain of assignable values. (The names of these domains stem from the sides of the assignment statement on which variables and assignable values are used: "left" and "right".) They are considered to be characteristic domains. Usually, as here, LV has Loc as a summand, and RV has SV as a summand.

The denotations of variable declarations and types are given in Table 2.23. It is convenient to introduce a second semantic function for variable declarations: for specifying that variables are no longer accessible—when exiting the scope of local variable declarations, for instance. Formally, the denotation of a variable declaration VD is the pair $(\mathcal{VD}[\![VD]\!], \mathcal{VU}[\![VD]\!])$.

The appropriate denotations for expressions, declarations, etc., are now functions of stores, as well as environments. Whether expression evaluation should be allowed to affect the store—known as "side-effects"—is controversial: some languages (such as C) actually encourage side-effects in expressions, but allow the order of evaluation of expressions to be specified; others make the order of evaluation of expressions "implementation-dependent", so that the semantics of programs that try to exploit side-effects in expressions becomes nondeterministic. Here, let us forbid side-effects, for simplicity. Thus denotations of expressions may be functions from environments and stores to expressible values—there is no need to return the current store, as it is unchanged.

We must modify the semantic equations for expressions, now that the denotations of expressions take stores as arguments. But first, note that in various contexts, there is an implicit "coercion" when the expression evaluation results in a variable, but the current value of the variable is required. Such contexts include operands of operators and conditions of if-then-else expressions. Very few

$$
\begin{array}{lcl}
\mathsf{S} & = & \mathsf{Loc} \to (\mathsf{SV} \oplus \mathsf{T}) \\
\mathsf{Loc} & = & \mathsf{O} \oplus \mathsf{Loc} \\
\textbf{empty} & = & \lambda l \in \mathsf{Loc}.\ \textbf{false} \\
 & \in & \mathsf{S} \\
\textbf{reservation} & = & \lambda l \in \mathsf{Loc}.\ \lambda s \in \mathsf{S}. \\
 & & (\lambda l' \in \mathsf{Loc}.\ \textbf{if}\ l =_{\mathsf{Loc}} l'\ \textbf{then true else}\ s(l')) \\
 & \in & \mathsf{Loc} \to \mathsf{S} \to \mathsf{S} \\
\textbf{freedom} & = & \lambda l \in \mathsf{Loc}.\ \lambda s \in \mathsf{S}. \\
 & & (\lambda l' \in \mathsf{Loc}.\ \textbf{if}\ l =_{\mathsf{Loc}} l'\ \textbf{then false else}\ s(l')) \\
 & \in & \mathsf{Loc} \to \mathsf{S} \to \mathsf{S} \\
\textbf{store} & = & \lambda l \in \mathsf{Loc}.\ \lambda v \in \mathsf{SV}.\ \lambda s \in \mathsf{S}. \\
 & & (\lambda l' \in \mathsf{Loc}.\ \textbf{if}\ l =_{\mathsf{Loc}} l'\ \textbf{then}\ v\ \textbf{else}\ s(l')) \\
 & \in & \mathsf{Loc} \to \mathsf{SV} \to \mathsf{S} \to \mathsf{S} \\
\textbf{stored} & = & \lambda l \in \mathsf{Loc}.\ \lambda s \in \mathsf{S}.\ [\mathsf{id}_{\mathsf{SV}}, \bot](s(l)) \\
 & \in & \mathsf{Loc} \to \mathsf{S} \to \mathsf{SV} \\
\textbf{location} & = & \textit{unspecified} \\
 & \in & \mathsf{S} \to \mathsf{Loc} \\
\textbf{allocation} & = & \lambda s \in \mathsf{S}.\ (\lambda l \in \mathsf{Loc}.\ (l, \textbf{reservation}\ l\ s))(\textbf{location}\ s) \\
 & \in & \mathsf{S} \to \mathsf{Loc} \times \mathsf{S}
\end{array}
$$

Table 2.21: Notation for stores

$$
\begin{array}{lcl}
\mathsf{LV} & = & \mathsf{Loc} \oplus \mathsf{LV}^* \\
\mathsf{RV} & = & \mathsf{SV} \oplus \mathsf{RV}^* \\
\\
\mathbf{allocations} & = & \lambda(f \in \mathsf{S} \to \mathsf{LV} \times \mathsf{S}, n \in \mathsf{N}_\perp). \\
 & & \mathbf{if}\ n = 0\ \mathbf{then}\ \lambda s \in \mathsf{S}.\ (\top, s)\ \mathbf{else} \\
 & & (\lambda(l \in \mathsf{LV}, s \in \mathsf{S}).\ (\lambda(l^* \in \mathsf{LV}^*, s' \in \mathsf{S}).\ ((l, l^*), s)) \\
 & & \qquad (\mathbf{allocations}(f, n-1)\, s)) \circ f \\
 & \in & (\mathsf{S} \to \mathsf{LV} \times \mathsf{S}) \times \mathsf{N}_\perp \to \mathsf{S} \to \mathsf{LV} \times \mathsf{S} \\
\mathbf{freedoms} & = & \lambda(f \in \mathsf{LV} \to \mathsf{S} \to \mathsf{S}, n \in \mathsf{N}_\perp). \\
 & & \mathbf{if}\ n = 0\ \mathbf{then}\ \lambda l^* \in \mathsf{O}.\ \mathsf{id}_\mathsf{S}\ \mathbf{else} \\
 & & \lambda(l \in \mathsf{LV}, l^* \in \mathsf{LV}^*).\ \mathbf{freedoms}(f, n-1)\, l^* \circ f\, l \\
 & \in & (\mathsf{LV} \to \mathsf{S} \to \mathsf{S}) \times \mathsf{N}_\perp \to \mathsf{LV} \to \mathsf{S} \to \mathsf{S} \\
\mathbf{component} & = & \lambda n \in \mathsf{N}_\perp.\ \mathbf{if}\ n = 1\ \mathbf{then}\ \mathbf{on}_1\ \mathbf{else}\ \mathbf{component}(n-1) \circ \mathbf{on}_2 \\
 & \in & \mathsf{N}_\perp \to \mathsf{LV}^* \to \mathsf{LV} \\
\mathbf{assign} & = & [\mathbf{store}, \\
 & & [\lambda l \in \mathsf{O}.\ \lambda v \in \mathsf{O}.\ \mathsf{id}_\mathsf{S}, \\
 & & \lambda(l \in \mathsf{LV}, l* \in \mathsf{LV}^*).\ \lambda(v \in \mathsf{RV}, v^* \in \mathsf{RV}^*). \\
 & & \qquad \mathbf{assign}\ l^*\ v^* \circ \mathbf{assign}\ l\ v]] \\
 & \in & \mathsf{LV} \to \mathsf{RV} \to \mathsf{S} \to \mathsf{S} \\
\mathbf{assigned} & = & [\mathbf{stored}, \\
 & & [\lambda l \in \mathsf{O}.\ \lambda s \in \mathsf{S}.\ (\top, s), \\
 & & \lambda(l \in \mathsf{LV}, l^* \in \mathsf{LV}^*).\ (\lambda(v \in \mathsf{RV}, s \in \mathsf{S}). \\
 & & \qquad (\lambda(v^* \in \mathsf{RV}^*, s' \in \mathsf{S}).\ ((v, v^*), s')) \\
 & & \qquad (\mathbf{assigned}\ l^*\ s)) \circ \mathbf{assigned}\ l]] \\
 & \in & \mathsf{LV} \to \mathsf{S} \to \mathsf{RV} \times \mathsf{S}
\end{array}
$$

Table 2.22: Notation for compound variables

$$\mathsf{SV} = \mathsf{T} \oplus \mathsf{Num}$$

$$\mathcal{VD} : \textsc{Variable-Declarations} \to \mathsf{S} \to (\mathsf{Env} \times \mathsf{S})$$

$$\mathcal{VD}[\![\ \texttt{var}\ I\texttt{:}\quad T\ ]\!] = (\lambda(l \in \mathsf{LV}, s \in \mathsf{S}).\ (\mathsf{binding}\ I\ l, s)) \circ \mathcal{T}[\![T]\!]$$

$$\mathcal{VD}[\![\ VD_1\texttt{,}\ \ VD_2\ ]\!] = (\lambda(e_1 \in \mathsf{Env}, s_1 \in \mathsf{S}).\ (\lambda(e_2 \in \mathsf{Env}, s_2 \in \mathsf{S}).\ (\mathsf{combine}(e_1, e_2), s_2))\ (\mathcal{VD}[\![VD_2]\!]s_1)) \circ \mathcal{VD}[\![VD_1]\!]$$

$$\mathcal{VU} : \textsc{Variable-Declarations} \to \mathsf{Env} \to \mathsf{S} \circ\!\!\to \mathsf{S}$$

$$\mathcal{VU}[\![\ \texttt{var}\ I\texttt{:}\quad T\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathcal{TU}[\![T]\!](\mathsf{bound}\ I\ e)$$

$$\mathcal{VU}[\![\ VD_1\texttt{,}\ \ VD_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathcal{VU}[\![VD_2]\!]e \circ \mathcal{VU}[\![VD_1]\!]e$$

$$\mathcal{T} : \textsc{Type} \to \mathsf{S} \to \mathsf{LV} \times \mathsf{S}$$

$$\mathcal{T}[\![\ \texttt{bool}\ ]\!] = \mathsf{allocation}$$

$$\mathcal{T}[\![\ \texttt{num}\ ]\!] = \mathsf{allocation}$$

$$\mathcal{T}[\![\ T\texttt{[1..}N\texttt{]}\ ]\!] = \mathsf{allocations}(\mathcal{T}[\![T]\!], \mathcal{N}[\![N]\!])$$

$$\mathcal{TU} : \textsc{Type} \to \mathsf{LV} \circ\!\!\to \mathsf{S} \circ\!\!\to \mathsf{S}$$

$$\mathcal{TU}[\![\ \texttt{bool}\ ]\!] = \mathsf{freedom}$$

$$\mathcal{TU}[\![\ \texttt{num}\ ]\!] = \mathsf{freedom}$$

$$\mathcal{TU}[\![\ T\texttt{[1..}N\texttt{]}\ ]\!] = \mathsf{freedoms}(\mathcal{TU}[\![T]\!], \mathcal{N}[\![N]\!])$$

Table 2.23: Denotations for variable declarations and types

programming languages insist that the programmer use an explicit operator on a variable in order to obtain its current value.

In practical programming languages, various coercions are allowed. A good example is the coercion from a parameterless function to the result of applying the function, allowed in ALGOL60 and PASCAL. Of course, a static semantic analysis could use contextual information to recognize such coercions and replace them by explicit operators. But in general, it is easy enough to deal with coercions directly in the dynamic semantics—although languages like ALGOL68 and ADA allow so many coercions that it may then be preferable to define the dynamic semantics on the basis of an intermediate abstract syntax where the coercions have been made explicit.

It is convenient to introduce a secondary semantic function for expressions, $\mathcal{R}$, that corresponds to ordinary evaluation followed by coercion (when possible). The modifications to our previous specification are straightforward; the result is shown in Table 2.24, together with the semantic equation for 'E_1 `[` E_2 `]`'.

2.5.6 Statements

The statements (or commands) of programming languages include *assignments* of values to variables, and constructs to control the order in which assignments are executed. Some typical syntax for statements is given in Table 2.25.

In the assignment statement 'E_1 `:=` E_2', the left-hand side E_1 must evaluate to a variable and E_2 must evaluate to an assignable value. The executions of the statements in 'S_1 `;` S_2' are sequenced (from left to right!) and '`skip`' corresponds to an empty sequence of statements. Conditional execution is provided by '`if` E `then` S_1', whereas '`while` E `do` S_1' iterates S_1 as long as E is true. The block '`begin` VD`;` S_1 `end`' limits the scope of the variable declarations in VD to the statements S_1, so that the variables themselves are "local" to the block, and may safely be re-used after the execution of S_1—assuming that "pointers" to local variables are not permitted. Let us defer consideration of the remaining statements in Table 2.25 until later in this section.

The denotational semantics of statements is quite simple: denotations are given by functions, from environments and stores, to stores. The bottom store represents the *non-termination* of statement execution, and the functions are strict in their store argument, reflecting that non-termination cannot be "ignored" by subsequent statements.

We are now ready to define the denotations of statements: see Table 2.26. Notice that the use of $\mathcal{VU}$ improves the abstractness of statement denotations: without it, the states produced by statement denotations would depend on the local variables allocated in inner blocks.

The following proposition is a direct consequence of the semantic equations, using the unfolding property of 'fix'.

Proposition 36

$$\mathcal{S}[\![\ \texttt{while}\ E\ \texttt{do}\ S_1\ ]\!] = \mathcal{S}[\![\ \texttt{if}\ E\ \texttt{then}\ (S_1\texttt{;}\ \texttt{while}\ E\ \texttt{do}\ S_1)\ ]\!].$$

$$F = (PV \multimap S \multimap FV)_\perp$$

$$PV = V \oplus F \oplus LV$$

$$FV = V$$

$$DV = V \oplus F \oplus LV$$

$$EV = V \oplus F \oplus LV$$

$\mathcal{R}$: EXPRESSION $\rightarrow$ Env $\rightarrow$ S $\rightarrow$ RV

$$\mathcal{R}[\![\ E\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ [\mathsf{id}_{\mathsf{RV}}, \perp, \lambda l \in \mathsf{LV}.\ \mathsf{assigned}\ l\ s](\mathcal{E}[\![E]\!] e\ s)$$

$\mathcal{E}$: EXPRESSION $\rightarrow$ Env $\rightarrow$ S $\rightarrow$ EV

$$\mathcal{E}[\![\ L\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ \mathcal{L}[\![L]\!]$$

$$\mathcal{E}[\![\ MO\ E_1\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ \mathcal{MO}[\![MO]\!](\mathcal{R}[\![E_1]\!] e\ s)$$

$$\mathcal{E}[\![\ E_1\ DO\ E_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ \mathcal{DO}[\![DO]\!](\mathsf{smash}(\mathcal{R}[\![E_1]\!] e\ s, \mathcal{R}[\![E_2]\!] e\ s))$$

$$\mathcal{E}[\![\ \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\ ]\!] =$$
$$\lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ (\lambda t \in \mathsf{T}.\ \mathsf{if}\ t\ \mathsf{then}\ \mathcal{E}[\![E_2]\!] e\ s\ \mathsf{else}\ \mathcal{E}[\![E_3]\!] e\ s)$$
$$(\mathcal{R}[\![E_1]\!] e\ s)$$

$$\mathcal{E}[\![\ I\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ \mathsf{bound}\ I\ e$$

$$\mathcal{E}[\![\ \texttt{let}\ CD\ \texttt{in}\ E\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ \mathcal{E}[\![E]\!](\mathsf{overlay}(\mathcal{CD}[\![CD]\!] e\ s, e)\ s)$$

$$\mathcal{E}[\![\ \texttt{fun}\ (PD)\ E\ ]\!] =$$
$$\lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ (\mathsf{up} \circ \mathsf{strict})(\lambda v \in \mathsf{PV}.\ \mathsf{id}_{\mathsf{FV}} \circ \mathcal{E}[\![E]\!](\mathsf{overlay}(\mathcal{PD}[\![PD]\!] v, e)))$$

$$\mathcal{E}[\![\ E_1(E_2)\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ (\mathsf{down} \circ \mathsf{id}_{\mathsf{F}})(\mathcal{E}[\![E_1]\!] e\ s)(\mathcal{E}[\![E_2]\!] e\ s)\ s$$

$$\mathcal{E}[\![\ E_1[E_2]\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ \mathsf{component}(\mathcal{E}[\![E_1]\!] e\ s, \mathcal{R}[\![E_2]\!] e\ s)$$

$\mathcal{CD}$: CONSTANT-DECLARATIONS $\rightarrow$ Env $\rightarrow$ S $\rightarrow$ Env

$$\mathcal{CD}[\![\ \texttt{val}\ I\ \texttt{=}\ E\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ \mathsf{binding}\ I\ (\mathcal{E}[\![E]\!] e\ s)$$

$$\mathcal{CD}[\![\ CD_1\ ;\ CD_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.$$
$$(\lambda e_1 \in \mathsf{Env}.\ \mathsf{overlay}(\mathcal{CD}[\![CD_2]\!](\mathsf{overlay}(e_1, e))\ s, e_1))$$
$$(\mathcal{CD}[\![CD_1]\!] e\ s)$$

$$\mathcal{CD}[\![\ \texttt{rec}\ CD\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ \mathsf{fix}(\lambda e' \in \mathsf{Env}.\ \mathcal{CD}[\![CD]\!](\mathsf{overlay}(e', e))\ s)$$

Table 2.24: Denotations for expressions (modified)

(STATEMENTS)

$$S ::= E_1 := E_2 \mid S_1; S_2 \mid \texttt{skip} \mid \texttt{if } E \texttt{ then } S_1 \mid \texttt{while } E \texttt{ do } S_1 \mid \texttt{begin } VD; S_1 \texttt{ end} \mid \texttt{stop} \mid I: S_1 \mid \texttt{goto } I$$

Table 2.25: Syntax for statements

$$\mathcal{S} : \text{STATEMENTS} \to \mathsf{Env} \to \mathsf{S} \multimap \mathsf{S}$$

$$\mathcal{S}[\![E_1 := E_2]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ (\lambda l \in \mathsf{LV}.\ \lambda v \in \mathsf{RV}.\ \mathsf{strict\ assign}\ l\ v\ s)\ (\mathcal{E}[\![E_1]\!]e\ s)(\mathcal{R}[\![E_2]\!]e\ s)$$

$$\mathcal{S}[\![S_1; S_2]\!] = \lambda e \in \mathsf{Env}.\ \mathcal{S}[\![S_2]\!]e \circ \mathcal{S}[\![S_1]\!]e$$

$$\mathcal{S}[\![\texttt{skip}]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{id_S}$$

$$\mathcal{S}[\![\texttt{if } E \texttt{ then } S_1]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ (\lambda t \in \mathsf{T}.\ \mathsf{if}\ t\ \mathsf{then}\ \mathcal{S}[\![S_1]\!]e\ s\ \mathsf{else}\ s)\ (\mathcal{R}[\![E]\!]e\ s)$$

$$\mathcal{S}[\![\texttt{while } E \texttt{ do } S_1]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{fix}(\lambda c \in \mathsf{S} \multimap \mathsf{S}.\ \lambda s \in \mathsf{S}.\ (\lambda t \in \mathsf{T}.\ \mathsf{if}\ t\ \mathsf{then}\ c(\mathcal{S}[\![S_1]\!]e\ s)\ \mathsf{else}\ s)\ (\mathcal{R}[\![E]\!]e\ s))$$

$$\mathcal{S}[\![\texttt{begin } VD; S_1 \texttt{ end}]\!] = \lambda e \in \mathsf{Env}.\ (\lambda(e' \in \mathsf{Env}, s \in \mathsf{S}).\ \mathcal{VU}[\![VD]\!](e')(\mathcal{S}[\![S_1]\!](\mathsf{overlay}(e', e))(s))) \circ \mathcal{VD}[\![VD]\!]e$$

Table 2.26: Denotations for statements (direct)

Now let us consider the statement '**stop**', whose intended effect is that when (if ever) the execution of a statement reaches it, the execution of the enclosing program is terminated—without further changes to the state, just as if control had reached the end of the program normally. We may say that '**stop**' causes a *jump* to the end of the program. (For now, let programs be simply statements. The semantics of programs is considered further in Section 2.5.8.)

However, with the denotations for statements used so far, we have (for any statement S_1 and $e \in \mathsf{Env}$):

$$\begin{aligned} \mathcal{S}[\![S_1; \texttt{while true do skip}]\!]e &= \mathsf{fix}(\mathsf{id}_{\mathsf{C}\to\mathsf{C}}) \circ \mathcal{S}[\![S_1]\!]e \\ &= \bot_{\mathsf{C}} \circ \mathcal{S}[\![S_1]\!]e = \bot_{\mathsf{C}} \end{aligned}$$

which is in conflict with the intended equivalence of '**stop; while true do skip**' to '**stop**'.

In order to deal with '**stop**', we clearly have to change the denotation of 'S_1; S_2'. There are two main techniques available for modeling jumps such as '**stop**': "flags", and "continuations".

The technique using *flags* is to use a domain of denotations such as $\mathsf{Env} \to \mathsf{S} \circ\!\!\to (\mathsf{S} \oplus \mathsf{S})$. Then a resulting store in (say) the first summand may represent normal termination, and a result in the second summand may represent that '**stop**' has been executed, so that no further statements are to be executed. Thus we would have

$$\mathcal{S}[\![\ S_1;\ S_2\ ]\!] = [\mathcal{S}[\![S_2]\!]e, \mathsf{in}_2] \circ (\mathcal{S}[\![S_1]\!]e)$$

It is easy to imagine the analogous changes that would be needed to the semantic equations for the other statements, to take account of the two possibilities for resulting stores. (No changes would be needed to the semantic equations for expressions and declarations, as they do not involve statements.)

The alternative technique for dealing with jumps is to let denotations of statements take *continuations* as arguments. The continuation argument represents the semantics of what would be the "rest of the program", if the statement were to terminate normally. In the denotation of each statement, it is specified whether to use the continuation argument, or to ignore it and use a different continuation, such as the empty continuation, which represents a jump to the end of the program. A divergent iterative statement just never gets around to using the argument continuation (and strictness is no longer needed to reflect the preservation of divergence).

In the rest of this section, the use of the continuations technique is illustrated, albeit briefly.

Let the characteristic domains (DV, EV, etc.) be as usual. The domain of *statement continuations* may be taken to be simply the domain $\mathsf{S} \to \mathsf{S}$ of functions on stores. For uniformity, let *all* denotations be functions of continuations. The continuations of expressions are functions from values to ordinary continuations, those for declarations are functions from environments to continuations, etc. (Auxiliary operations, such as **assign**, could be changed to take continuation arguments as well, if desired.) Such a semantics is called a "continuation semantics"; our previous examples of semantics are called "direct".

Sufficient semantic equations to illustrate the technique of continuations are given in Table 2.27. (The semantic functions of the continuation semantics are marked with primes to distinguish them from the corresponding direct semantic functions.) Notice the order of composition in the semantic equation for 'S_1; S_2': the opposite to that in direct semantics!

The transformation from direct to continuation semantics is straightforward. It may seem quite obvious that the transformation gives an "equivalent" semantics, but it is non-trivial to prove such results: the relations to be established between the domains of the direct and continuation semantics have to be defined recursively, and then shown to be well-defined and "inclusive" [44].

Continuations were originally introduced to model the semantics of general '**goto**'-statements. Consider again the syntax given in Table 2.25. An occurrence of a labeled statement 'I: S_1' may be regarded as a *declaration* that binds I, where the scope of this binding is the smallest enclosing block '**begin** VD; S_1 **end**'.

The execution of '**goto** I' is intended to jump to the statement labeled by I. It may be seen to consist of

$$\mathsf{C} = \mathsf{S} \to \mathsf{S}$$

$$\mathcal{E}' : \textsc{Expression} \to \mathsf{Env} \to (\mathsf{EV} \to \mathsf{C}) \to \mathsf{C}$$

$$\mathcal{R}' : \textsc{Expression} \to \mathsf{Env} \to (\mathsf{RV} \to \mathsf{C}) \to \mathsf{C}$$

$$\mathcal{VD}' : \textsc{Variable-Declarations} \to \mathsf{Env} \to (\mathsf{Env} \to \mathsf{C}) \to \mathsf{C}$$

$$\ldots$$

$$\mathcal{S}' : \textsc{Statements} \to \mathsf{Env} \to \mathsf{C} \to \mathsf{C}$$

$$\mathcal{S}'[\![\ E_1 \ \texttt{:=}\ E_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda c \in \mathsf{C}.\ \mathcal{E}'[\![E_1]\!]e\,(\lambda l \in \mathsf{LV}.\ \mathcal{R}'[\![E_2]\!]e\,(\lambda v \in \mathsf{RV}.\ c \circ (\mathsf{assign}\ l\ v)))$$

$$\mathcal{S}'[\![\ S_1\texttt{;}\ S_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda c \in \mathsf{C}.\ \mathcal{S}'[\![S_1]\!]e\,(\mathcal{S}'[\![S_2]\!]e\,c)$$

$$\mathcal{S}'[\![\ \texttt{skip}\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda c \in \mathsf{C}.\ c$$

$$\mathcal{S}'[\![\ \texttt{while}\ E\ \texttt{do}\ S_1\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{fix}(\lambda g \in \mathsf{C} \to \mathsf{C}.\ \lambda c \in \mathsf{C}.\ \mathcal{R}'[\![E]\!]e\,(\lambda t \in \mathsf{T}.\ \mathsf{if}\ t\ \mathsf{then}\ \mathcal{S}'[\![S_1]\!]e\,(g(c))\ \mathsf{else}\ c))$$

$$\mathcal{S}'[\![\ \texttt{stop}\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda c \in \mathsf{C}.\ \mathsf{id}_{\mathsf{S}}$$

$$\mathcal{S}'[\![\ \texttt{goto}\ I\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda c \in \mathsf{C}.\ \mathsf{bound}\ I\ e$$

$$\mathcal{LD}' : \textsc{Statements} \to \mathsf{Env} \to \mathsf{C} \to \mathsf{Env}$$

Table 2.27: Denotations for statements (continuations)

1. the termination of enclosing statements (including procedure calls) up to the innermost '**begin** VD; S_1 **end**' that includes the declaration of the label I; then

2. the execution of those parts of S_1 that follow after the label I; and finally

3. the normal termination of '**begin** VD; S_1 **end**', provided that no further jump prevents this.

(Actually, this analysis suggests a direct semantics using flags, where label identifiers are bound to pairs consisting of "activation levels" and direct statement denotations: continuations are not actually *necessary* for the denotational description of '**goto**'-statements.)

Letting C be a summand of DV, the value bound to I by 'I: S_1' is $\mathcal{S}'[\![S_1]\!]e\,c$, where c is the continuation argument of $\mathcal{S}'[\![\ I\texttt{:}\ S_1\ ]\!]e$. So assuming that the environment argument e includes this binding, the denotation of the '**goto**'-statement merely replaces its argument continuation by the continuation bound to I, as specified in Table 2.27. The declarative component of statement

denotations may be expressed by a semantic function $\mathcal{LD}'$ whose definition involves a fixed point, which reflects that the continuations denoted by label identifiers in a block may be mutually-recursive. The details are somewhat tedious; let us omit them here, as unrestricted jumps to labels are not allowed in most modern high-level programming languages.

Note that continuations give possibilities for jumps that are even less "disciplined" than those provided by the 'goto' statement: a general continuation need have no relation at all to the context of where it is used!

Continuations have been advocated as a standard technique for modeling programming languages (along with the use of environments and states) in preference to direct semantics. Although the adoption of this policy would give a welcome uniformity in models, it would also make the domains of denotations for simple languages (e.g., the λ-calculus) unnecessarily complex—and, at least in some cases, the introduction of continuations would actually reduce the abstractness of denotations.

The popularity of continuations seems to be partly due to the accompanying notational convenience—especially that the order in which denotations of sub-phrases occur in semantic equations corresponds to the order in which the phrases are intended to be executed: left to right. (Perhaps direct semantics would be more popular if function application and composition were to be written "backwards".) Another notational virtue of continuations is that "errors" can be handled neatly, by ignoring the continuation argument and using a general error-continuation.

2.5.7 Procedure Abstractions

Procedure abstractions are much like function abstractions. The only difference is that the body of a procedure abstraction is a statement, rather than an expression.

By the way, many programming languages do not allow functions to be expressed (or declared) directly: procedures must be used instead. The body of the procedure then includes a special statement that determines the value to be returned (in ALGOL60 and PASCAL, this statement looks like an assignment to the procedure identifier!).

Syntax for procedure abstractions is given in Table 2.28. As with functions, we consider procedures with only a single parameter; but now some more modes of parameter evaluation are introduced.

(EXPRESSION)
E ::= **proc** (PD) S_1

(PARAMETER-DECLARATION)
PD ::= **var** I: T | I: T

(STATEMENTS)
S ::= $E_1(E_2)$

Table 2.28: Syntax for procedures

The procedure abstraction '**proc** (**var** I: T) S_1' requires its parameter to evaluate to a variable, and I denotes that variable in the body S_1. This mode of parameter evaluation is usually known as "call by reference", but here we refer to it as "variable-mode" parameter evaluation.

The procedure abstraction '**proc** (I: T) S_1' requires its parameter to be *coercible* to an assignable value; then a local variable is allocated and initialized with the parameter value, and I denotes the variable in the body S_1. This mode of parameter evaluation is usually known as "call by value", but it should not be confused with the value-mode parameter evaluation that was considered for function abstractions: that did not involve any local variable allocation. Let us refer to this mode as "copy-mode" parameter evaluation.

The procedure call statement '$E_1(E_2)$' executes the body of the procedure abstraction produced by evaluating E_1, passing the argument obtained by evaluating the parameter E_2.

Note that execution of the procedure body may have an effect on the state, by assignment to a non-local variable. With variable-mode parameters, there is also the possibility of modifying the state by assigning to the formal parameter of the abstraction; whereas with copy-mode, such an assignment merely modifies the *local* variable denoted by the parameter identifier. Note also that variable-mode allows two different identifiers to denote the same variable, i.e., "aliasing".

Now for the formal semantics of procedures. The denotations of procedure expressions, parameter declarations, and statements are defined in Table 2.29.

The procedure call syntax '$E_1(E_2)$' does not give any indication of the mode of parameter evaluation, so we leave it to the denotation of the parameter declaration to perform any required coercion of the parameter value. An alternative technique is to let the evaluation of the parameter expression E_2 depend on a mode component of the value of the procedure expression E_1.

By the way, the second semantic function for parameter declarations, $\mathcal{PU}$, is analogous to the semantic function $\mathcal{VU}$ for variable declarations, explained in Section 2.5.5.

2.5.8 Programs

As discussed in Section 2.3, the semantics of an entire program should be a mathematical representation of the observable behaviour when it is executed by computers (but ignoring implementation-dependent details). Typically, this behaviour involves *streams* of "input" and "output".

By definition, the *input* of a program is the information that is supplied to it by the user; the *output* is the information that the user gets back. However, it is important to take into account not only *what* information is supplied, but also *when* the supply takes place. The main distinction in conventional programming languages is between so-called "batch" and "interactive" input-output.

With *batch* input, all the input to the program is supplied at the start of the program. The input may then be regarded as *stored*, in a "file". Batch output is likewise accumulated in a file, and only given to the user when (if ever) the program terminates.

On the other hand, *interactive* input is provided gradually, as a *stream* of data, while the program is running; the program may have to wait for further input data to be provided before it can proceed. Similarly, interactive output is provided to the user while the program is running, as

$$\mathsf{P} = (\mathsf{PV} \multimap \mathsf{S} \multimap \mathsf{S})_\perp$$

$$\mathsf{PV} = \mathsf{V} \oplus \mathsf{F} \oplus \mathsf{LV} \oplus \mathsf{P}$$

$$\mathsf{DV} = \mathsf{V} \oplus \mathsf{F} \oplus \mathsf{LV} \oplus \mathsf{P}$$

$$\mathsf{EV} = \mathsf{V} \oplus \mathsf{F} \oplus \mathsf{LV} \oplus \mathsf{P}$$

$$\mathcal{E} : \textsc{Expression} \to \mathsf{Env} \to \mathsf{S} \to \mathsf{EV}$$

$$\mathcal{E}[\![\ \texttt{proc}\ (PD)\ S\ ]\!] =$$
$$\lambda e \in \mathsf{Env}.\ \mathsf{up}(\mathsf{strict}\lambda v \in \mathsf{PV}.$$
$$(\lambda(e' \in \mathsf{Env}, s \in \mathsf{S}).\ \mathcal{PU}[\![PD]\!]e'(\mathcal{S}[\![S]\!](\mathsf{overlay}(e', e))))$$
$$\circ\, \mathcal{PD}[\![PD]\!]e)$$

$$\mathcal{PD} : \textsc{Parameter-Declaration} \to \mathsf{PV} \to \mathsf{S} \to \mathsf{Env} \times \mathsf{S}$$

$$\mathcal{PD}[\![\ \texttt{val}\ I\colon\ \ T\ ]\!] = \lambda v \in \mathsf{PV}.\ \lambda s \in \mathsf{S}.\ (\mathsf{binding}\ I\, v, s)$$

$$\mathcal{PD}[\![\ \texttt{var}\ I\colon\ \ T\ ]\!] = \lambda l \in \mathsf{LV}.\ \lambda s \in \mathsf{S}.\ (\mathsf{binding}\ I\, l, s)$$

$$\mathcal{PD}[\![\ I\colon\ \ T\ ]\!] = \lambda v \in \mathsf{PV}.\ \lambda s \in \mathsf{S}.$$
$$(\lambda v' \in \mathsf{RV}.\ (\lambda(l' \in \mathsf{LV}, s' \in \mathsf{S}).\ (\mathsf{binding}\ I\, l', \mathsf{assign}\ l'\, v'\, s'))$$
$$(\mathcal{T}[\![T]\!]s))$$
$$([\lambda l \in \mathsf{LV}.\ \mathsf{assigned}\ l\, s, \mathsf{id}_{\mathsf{RV}}, \perp, \perp](v))$$

$$\mathcal{PU} : \textsc{Parameter-Declaration} \to \mathsf{Env} \to \mathsf{S} \to \mathsf{S}$$

$$\mathcal{PU}[\![\ \texttt{val}\ I\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{id}_{\mathsf{S}}$$

$$\mathcal{PU}[\![\ \texttt{var}\ I\colon\ \ T\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{id}_{\mathsf{S}}$$

$$\mathcal{PU}[\![\ I\colon\ \ T\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathcal{TU}[\![T]\!](\mathsf{bound}\ I\, e)$$

$$\mathcal{S} : \textsc{Statements} \to \mathsf{Env} \to \mathsf{S} \multimap \mathsf{S}$$

$$\mathcal{S}[\![\ E_1(E_2)\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ (\mathsf{down} \circ \mathsf{id}_{\mathsf{P}})(\mathcal{E}[\![E_1]\!]e\, s)(\mathcal{E}[\![E_2]\!]e\, s)\, s$$

Table 2.29: Denotations for procedures

soon as it has been determined.

Note that interactive input-output allows (later) items of input to *depend* on (earlier) items of output. For instance, input may be stimulated by an output "prompt".

We may regard batch input-output as merely a special case of interactive input-output: the program starts, and then immediately reads and stores the entire input; output is stored until the program is about to terminate, and then the entire output is given to the user.

The *essential* difference between batch and interactive input-output shows up in connection with programs that (on purpose) may run "for ever": batch input-output cannot reflect the semantics of such programs. Familiar examples are traffic-light controllers, operating systems, and screen editors. These programs might, if allowed, read an infinite stream of input, and produce an infinite stream of output. (They might also terminate, in response to particular input—or "spontaneously", when an error occurs.) Moreover, once an item of output has been produced, it cannot be revoked by the program (e.g., the traffic-light controller cannot "undo" the changing of a light).

Consider the abstract syntax for input-output statements and programs specified in Table 2.30. There is nothing in the given *syntax* that indicates whether the *semantics* of input-output is supposed to be batch or interactive. Let us consider both semantics. We restrict items of input and output to be truth-values and numbers, i.e., the same as SV.

(PROGRAM)

P ::= **prog** S

(STATEMENTS)

S ::= **read** E | **write** E

Table 2.30: Syntax for programs

For batch semantics, we may take the representation of streams to be finite lists. The semantic equations for programs, and for read and write statements, are given in Table 2.31; our previous semantic equations for other statements have to be modified to take account of the extra arguments, but the details are omitted here.

The following proposition confirms that batch output is not observable when program execution doesn't terminate:

Proposition 37

$$\mathcal{P}[\![\ \texttt{prog while true do write 0}\ ]\!] = \mathcal{P}[\![\ \texttt{prog while true do skip}\ ]\!] = \bot.$$

The reason for this is that the denotation of the non-terminating while-loop is given by the *least* fixed point of a *strict* function.

Now for interactive input-output semantics for the same language. See Table 2.32. Let us first change from SV^* to $\mathsf{SV}^\$$, which represents infinite (and partial) streams. (The only difference between $\mathsf{SV}^\$$ and the standard domain construction SV^∞ is that the latter allows $\bot$ components to be followed by non-$\bot$ components.) This change by itself would *not* make any substantial difference

$$\mathsf{In} = \mathsf{SV}^*$$

$$\mathsf{Out} = \mathsf{SV}^*$$

$$\mathcal{P} : \textsc{Program} \to (\mathsf{In} \multimap \mathsf{Out})$$

$$\mathcal{P}[\![\ \texttt{prog}\ S_1\ ]\!] = \lambda i \in \mathsf{In}.\ \mathsf{on}_3(\mathcal{S}[\![S_1]\!](\mathsf{void})(\mathsf{empty}, i, \top))$$

$$\mathcal{S} : \textsc{Statements} \to \mathsf{Env} \to (\mathsf{S} \otimes \mathsf{In} \otimes \mathsf{Out}) \multimap (\mathsf{S} \otimes \mathsf{In} \otimes \mathsf{Out})$$

$$\begin{aligned}\mathcal{S}[\![\ \texttt{read}\ E\ ]\!] = {} & \lambda e \in \mathsf{Env}.\ \lambda(s \in \mathsf{S}, i \in \mathsf{In}, o \in \mathsf{Out}).\\ & (\lambda l \in \mathsf{Loc}.\ [\bot, \lambda(v \in \mathsf{SV}, i' \in \mathsf{In}).\ \mathsf{smash}(\mathsf{store}\ l\, v\, s, i', o)])\\ & (\mathcal{E}[\![E]\!] e\, s)(i)\end{aligned}$$

$$\begin{aligned}\mathcal{S}[\![\ \texttt{write}\ E\ ]\!] = {} & \lambda e \in \mathsf{Env}.\ \lambda(s \in \mathsf{S}, i \in \mathsf{In}, o \in \mathsf{Out}).\\ & (\lambda v \in \mathsf{SV}.\ \mathsf{smash}(s, i, \mathsf{extend}\ v\, o))\\ & (\mathcal{R}[\![E]\!] e\, s)\end{aligned}$$

$$\begin{aligned}\mathsf{extend} &= \lambda v \in \mathsf{SV}.\ [\lambda x \in \mathsf{O}.\ (v, \top),\\ &\qquad \lambda(v' \in \mathsf{SV}, o \in \mathsf{Out}).\ (v', \mathsf{extend}\ v\, o)]\\ &\in \mathsf{SV} \to \mathsf{Out} \to \mathsf{Out}\end{aligned}$$

Table 2.31: Denotations for programs (batch)

to the semantics of programs: input-output would still be batch, and the above proposition would still hold.

The essential change is to ensure that an item of output becomes incorporated in the program's semantics, irrevocably, as soon as the corresponding '`write`' statement is executed. There are various ways of achieving this property: in particular, by using continuations. Reverting temporarily to continuation semantics (see Section 2.5.6) we define the interactive semantics of programs as shown in Table 2.32.

Proposition 38

$$\mathcal{P}'[\![\ \texttt{prog while true do write 0}\ ]\!] \neq$$

$$\mathcal{P}'[\![\ \texttt{prog while true do skip}\ ]\!].$$

It is instructive to see how to deal with interactive input-output without using continuations. Consider the domain IO defined in Table 2.33, and let statement denotations be given by functions from environments and stores to IO. Each element of IO represents a sequence of readings and

$$\mathsf{SV}^\$ = \mathsf{SV} \otimes \mathsf{SV}^\$_\perp$$

$$\mathsf{In} = \mathsf{SV}^\$$$

$$\mathsf{Out} = \mathsf{SV}^\$$$

$$\mathsf{C} = \mathsf{S} \to \mathsf{In} \to \mathsf{Out}$$

$$\mathcal{P}' : \textsc{Program} \to \mathsf{In} \to \mathsf{Out}$$

$$\mathcal{P}'[\![\ \texttt{prog}\ S_1\ ]\!] = \mathcal{S}'[\![S_1]\!](\mathsf{void})(\lambda s \in \mathsf{S}.\ \lambda i \in \mathsf{In}.\ \top)(\mathsf{empty}).$$

$$\mathcal{S}' : \textsc{Statements} \to \mathsf{Env} \to \mathsf{C} \to \mathsf{C}$$

$$\mathcal{S}'[\![\ \texttt{read}\ E\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda c \in \mathsf{C}.\ \mathcal{E}'[\![E]\!]e\,(\lambda l \in \mathsf{Loc}.\ \lambda s \in \mathsf{S}.\ \lambda(v \in \mathsf{SV}, i \in \mathsf{In}).\ (c \circ \mathsf{store}\ l\ v)\ s\ i))$$

$$\mathcal{S}'[\![\ \texttt{write}\ E\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda c \in \mathsf{C}.\ \mathcal{R}'[\![E]\!]e(\lambda v \in \mathsf{SV}.\ \lambda s \in \mathsf{S}.\ \lambda i \in \mathsf{In}.\ \mathsf{smash}(v, \mathsf{up}(c\,s\,i)))$$

Table 2.32: Denotations for programs (interactive, continuations)

writings, ending (if at all) with a state. This might not seem particularly abstract, but notice that statement denotations *must* reflect the order in which readings and writing occur, since the semantics of a program in $\mathsf{In} \to \mathsf{Out}$ reveals this information when applied to *partial* inputs.

The semantic equations specified in Table 2.33 illustrate this technique. The fixed point used in the denotation of 'S_1 ; S_2' essentially corresponds to going through the input and output corresponding to S_1 until a final state is reached, and then starting S_2; similarly for programs. It can be shown that interactive output is modeled.

Now consider "piping" the output of one program into the input of another, as expressed by a program construct 'P_1 | P_2'. With interactive input-output, both programs can be started simultaneously—but the execution of the second program may have to be suspended to await input that has yet to be output by the first program. The start of the first program could be delayed until the second program actually tries to read from its input (if ever), and then execution could alternate between the two programs, according to the input-output. All these possibilities are expressed by the same semantic equation:

$$\mathcal{P}[\![\ P_1\ |\ P_2\ ]\!] = \mathcal{P}[\![P_2]\!] \circ \mathcal{P}[\![P_1]\!].$$

With batch input-output, the second program does not start until the first one terminates. As with statements, such sequential execution can be modeled by composition of strict functions

$$\mathsf{IO} = \mathsf{S} \oplus (\mathsf{SV} \multimap \mathsf{IO})_\perp \oplus (\mathsf{SV} \otimes \mathsf{IO}_\perp).$$

$\mathcal{P}$: PROGRAM $\to$ In $\to$ Out

$$\begin{aligned}
\mathcal{P}[\![\ \texttt{prog}\ S_1\ ]\!] = \mathsf{fix}(&\lambda h \in \mathsf{IO} \to \mathsf{In} \to \mathsf{Out}.\\
&[\lambda s \in \mathsf{S}.\ \lambda i \in \mathsf{In}.\top,\\
&\ \lambda f \in \mathsf{SV} \to \mathsf{IO}.\ \lambda(v \in \mathsf{SV}, i \in \mathsf{In}).\ h(f(v))(i),\\
&\ \lambda(v \in \mathsf{SV}, io \in \mathsf{IO}).\ \lambda i \in \mathsf{In}.\ (v, \mathsf{up}(h(io)(i)))])\\
&(\mathcal{S}[\![S_1]\!](\mathsf{void})(\mathsf{empty}))
\end{aligned}$$

$\mathcal{S}$: STATEMENTS $\to$ Env $\to$ S $\multimap$ IO

$$\begin{aligned}
\mathcal{S}[\![\ E_1\ \texttt{:=}\ E_2\ ]\!] &= \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\\
&\quad (\lambda l \in \mathsf{Loc}.\ \lambda v \in \mathsf{SV}.\ \mathsf{in}_1(\mathsf{store}\ l\ v\ s))\\
&\quad (\mathcal{E}[\![E_1]\!]e\ s)(\mathcal{R}[\![E_2]\!]e\ s)\\
\mathcal{S}[\![\ \texttt{read}\ E\ ]\!] &= \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\\
&\quad (\lambda l \in \mathsf{Loc}.\ \mathsf{in}_2(\lambda v \in \mathsf{SV}.\ \mathsf{up}(\mathsf{in}_1(\mathsf{store}\ l\ v\ s))))\\
&\quad (\mathcal{E}[\![E_1]\!]e\ s)\\
\mathcal{S}[\![\ \texttt{write}\ E\ ]\!] &= \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\\
&\quad (\lambda v \in \mathsf{SV}.\ \mathsf{in}_3(v, \mathsf{up}(\mathsf{in}_1(s))))\\
&\quad (\mathcal{R}[\![E_2]\!]e\ s)\\
\mathcal{S}[\![\ \texttt{skip}\ ]\!] &= \lambda e \in \mathsf{Env}.\ \mathsf{id}_\mathsf{S}\\
\mathcal{S}[\![\ S_1\texttt{;}\ S_2\ ]\!] &= \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\\
&\quad \mathsf{fix}(\lambda g \in \mathsf{IO} \to \mathsf{IO}.\\
&\qquad [\mathcal{S}[\![S_2]\!]e,\\
&\qquad\ \lambda f \in \mathsf{SV} \to \mathsf{IO}.\ g \circ f,\\
&\qquad\ \lambda(v \in \mathsf{SV}, io \in \mathsf{IO}).\ (v, \mathsf{up}(g(io)))])\\
&\quad (\mathcal{S}[\![S_1]\!]e\ s)
\end{aligned}$$

Table 2.33: Denotations for programs (interactive, direct)

(the semantic equation for piped programs remains the same, assuming $\mathcal{P}$ is defined as for batch input-output).

2.5.9 Nondeterminism

The final technique illustrated in this chapter is the use of *power domains* to model nondeterministic constructs such as "guarded commands" and interleaving.

For our purposes here, it is not necessary to understand the actual structure of power domains. All that we need to know about a power domain is that it is equipped with a continuous union operation (associative, commutative, and absorptive), a continuous singleton operation, and that functions on domains can be extended pointwise to power domains. (Recall the notation adopted in Section 2.4. We use only the natural, or convex, power domain; the other power domains do not accurately reflect the possibility of divergence.)

Consider the syntax for *guarded statements* given in Table 2.34. The intention of 'E -> S_1' is that the statement S_1 is guarded by E and may only be executed if E evaluates to true. So far, this resembles '**if** E **then** S_1'; the difference is that guarded statements may be "united" by the construct 'G_1 [] G_2', whose execution consists of executing precisely one of the guarded statements in G_1 and G_2. Notice that (when E evaluates to a truth-value) the guarded statement

$$E \texttt{ -> } S_1 \texttt{ [] } \neg E \texttt{ -> } S_2$$

expresses a deterministic choice between S_1 and S_2, whereas

$$\texttt{true -> } S_1 \texttt{ [] true -> } S_2$$

expresses a nondeterministic choice.

(GUARDED-STATEMENTS)

G ::= E -> S | G_1 [] G_2

(STATEMENTS)

S ::= **if** G **fi** | **do** G **od**

Table 2.34: Syntax for guarded statements

Both the statements '**if** G **fi**' and '**do** G **od**' involve the execution of G, when possible. Let us regard the former as equivalent to an empty statement when it is not possible to execute G. With the latter, the execution of G is *repeated*, as many times as possible.

We take the denotations for statements to be functions from environments and stores to elements of the power domain $\mathsf{S}^\natural$; these elements represent the non-empty sets of possible states resulting from statement execution (possibly including $\perp$). The denotations of guarded statements are similar, but $\top$ represents the empty set of states. The semantic equations are specified in Table 2.35. (We do not need to change the denotations of expressions and declarations, which are still deterministic.)

$$\mathcal{G} : \text{GUARDED-STATEMENTS} \to \mathsf{Env} \to \mathsf{S} \multimap (\mathsf{O} \oplus \mathsf{S}^\natural)$$

$$\mathcal{G}[\![\ E \texttt{ -> } S_1\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$(\lambda t \in \mathsf{T}.\ \mathsf{if}\ t\ \mathsf{then}\ \mathsf{in}_2(\mathcal{S}[\![S_1]\!]e\,s)\ \mathsf{else}\ \mathsf{in}_1\,\top)(\mathcal{R}[\![E]\!]e\,s)$$

$$\mathcal{G}[\![\ G_1 \ \square\ G_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$[\lambda x \in \mathsf{O}.\ \mathsf{id}_{\mathsf{O}\oplus\mathsf{S}^\natural},$$
$$\lambda p_1 \in \mathsf{S}^\natural.\ [\lambda x \in \mathsf{O}.\ \mathsf{in}_2(p_1),$$
$$\lambda p_2 \in \mathsf{S}^\natural.\ \mathsf{in}_2(p_1 \uplus p_2)]](\mathcal{G}[\![G_1]\!]e\,s)(\mathcal{G}[\![G_2]\!]e\,s)$$

$$\mathcal{S} : \text{STATEMENTS} \to \mathsf{Env} \to \mathsf{S} \multimap \mathsf{S}^\natural$$

$$\mathcal{S}[\![\ \texttt{if}\ G\ \texttt{fi}\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$[\lambda x \in \mathsf{O}.\ \{\!|s|\!\}, \mathsf{id}_{\mathsf{S}^\natural}](\mathcal{G}[\![G]\!]e\,s)$$

$$\mathcal{S}[\![\ \texttt{do}\ G\ \texttt{od}\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{fix}(\lambda c \in \mathsf{S} \multimap \mathsf{S}^\natural.\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$[\lambda x \in \mathsf{O}.\ \{\!|s|\!\}, \mathsf{ext}(c)](\mathcal{G}[\![G]\!]e\,s))$$

Table 2.35: Denotations for guarded statements

As an illustration of the semantic equivalence that is induced by the above definitions, consider the two statements S_1, S_2 shown in Table 2.36. It is obvious that S_2 has the possibility of not terminating; what may be less obvious is that S_1 has precisely the same possibilities:

Proposition 39

$$\mathcal{S}[\![S_1]\!] = \mathcal{S}[\![S_2]\!].$$

Thus both statements have the possibility of terminating with the variable '`y`' having *any* (non-negative) value—or of not terminating. The infinite number of possibilities arises here from the *iteration* of a choice between a *finite* number of possibilities: the possibility of non-termination cannot be eliminated (c.f. König's Lemma).

```
x := 0;                          x := 0;
y := 0;                          y := 0;
do x=0  -> x := 1                do x=0  -> x := 1
[] x=0  -> y := y+1              [] x=0  -> y := y+1
od                               [] true -> do true -> skip od
                                 od
```

Table 2.36: Examples of guarded statements S_1, S_2

However, one could imagine having a *primitive* statement with an infinite number of possibilities, excluding non-termination. E.g., consider '`randomize` E', which is supposed to set a variable E

to some arbitrary integer. Here we understand "arbitrary" to mean just that the value chosen is completely out of the control of the program—it is implementation-dependent. (Thus a particular implementation might always choose zero, or the successor of the previous choice. Classes of genuinely-random implementations could be considered as well.)

It is important to note that our domain of statement denotations above does *not* contain any element that can be used for the denotation of an always-terminating '`randomize`' statement. In fact any attempt to express such a set as

$$\{0\} \cup \{1\} \ldots \{n\} \ldots$$

as an element of $\mathsf{N}_\perp^\natural$ always ends up by including $\{\perp\}$ as well.

So let us omit further consideration of randomizing statements, and proceed to illustrate a technique known as "resumptions", which is useful for giving a denotational semantics for concurrent processes.

2.5.10 Concurrency

The language constructs considered so far in this chapter come from conventional programming languages, designed to be implemented *sequentially*. Several modern programming languages have constructs for expressing so-called "concurrent processes", and may be implemented on a "distributed system" of computers (or on a single computer that simulates a distributed system). Typically, the processes are executed asynchronously, and they interact by sending messages and making "rendezvous".

In the denotational semantics of concurrent systems, the concurrent execution steps of different processes are usually regarded as "interleaved". Although interleaving is a rather artificial concept when dealing with physically-distributed systems (due to the lack of a universal time scale) it is not generally possible to distinguish the possible behaviours of proper concurrent systems from their interleaved counterparts—at least, not unless the observer of the behaviours is distributed too.

The final example of this chapter deals with a very simple form of concurrency: interleaved statements. The syntax of these statements is given in Table 2.37.

(STATEMENTS)
$S \;::=\; S_1 \;\|\|\; S_2 \;\mid\; < S >$

Table 2.37: Syntax for interleaved statements

The intention with the statement 'S_1 || S_2' is that S_1 and S_2 are executed concurrently and asynchronously. If S_1 and S_2 use the same variables, the result of their concurrent execution may depend on the order in which the "steps" of S_1 are executed in relation to those of S_2, i.e., on the interleaving. Let us assume that assignment statements are single, "indivisible" steps of execution, so the state does not change during the evaluation of the left- and right-hand sides. The construct '< S_1 >' makes the execution of any statement S_1 an indivisible step (sometimes called a "critical region").

Note that when S_1 and S_2 are "independent" (e.g., when they use different variables) an execution of 'S_1 || S_2' gives the same result as the execution of 'S_1 ; S_2', or of 'S_2 ; S_1'; but in general there are other possible results.

Now consider statements

S_1 : `x := 1`

S_2 : `x := 0; x := x+1.`

With all our previous denotations for statements, we have $\mathcal{S}[\![S_1]\!] = \mathcal{S}[\![S_2]\!]$. But when statements include 'S_1 || S_2', we expect

$$\mathcal{S}[\![\ S_1\ ||\ S_1\ ]\!] \neq \mathcal{S}[\![\ S_1\ ||\ S_2\ ]\!]$$

since the interleaving '`x := 0; x := 1; x := x+1`' of S_1 with S_2 sets `x` to 2, whereas the interleaving of '`x := 1`' with itself does not have this possibility.

Thus it can be seen that the compositionality of denotational semantics forces $\mathcal{S}[\![S_1]\!] \neq \mathcal{S}[\![S_2]\!]$ when concurrent statements are included. The appropriate denotations for statements are so-called "resumptions", which are rather like segmented ("staccato") continuations. A domain of resumptions is defined in Table 2.38. The semantic function for statements, $\mathcal{S}$, maps environment directly to resumptions, which are themselves functions of stores.

Consider $p = \mathcal{S}[\![S_1]\!]e\,s$. It represents the set of possible results of executing the *first step* of S_1. An element $\mathsf{in}_1(s')$ of this set corresponds to the possibility that there is only one step, resulting in the state s' (although this "step" might be an indivisible sequence of steps). An element $\mathsf{in}_2(\mathsf{up}\,r, s')$ corresponds to the result of the first step being an *intermediate* state s', together with a resumption r which, when applied to s' (or to some other state) gives the set of possible results from the next step of S_1, and so on.

Resumptions provide adequate denotations for interleaved statements, as the semantic equations in Table 2.38 show. However, these denotations are not particularly abstract: e.g., we get $\mathcal{S}[\![$ `skip` $]\!] \neq \mathcal{S}[\![$ `skip; skip` $]\!]$, even though the two statements are clearly interchangeable in any program. It is currently an open problem to define fully abstract denotations for concurrent interleaved statements (using standard semantic domain constructions).

The technique of resumptions can also be used for expressing denotations of *communicating* concurrent processes (with the "store" component representing pending communications).

We have finished illustrating the use of the main descriptive techniques of Denotational Semantics: environments, stores, strictness, flags, continuations, power domains, and resumptions. The various works referenced in the following bibliographical notes provide further illustrations of the use of these techniques, and show how to obtain denotations for many of the constructs to be found in "real" programming languages.

$$\mathsf{R} = \mathsf{S} \circ\!\!\rightarrow (\mathsf{S} \oplus (\mathsf{R}_\perp \otimes \mathsf{S}))^\natural$$

$$\mathcal{S} : \textsc{Statements} \rightarrow \mathsf{Env} \rightarrow \mathsf{R}$$

$$\mathcal{S}[\![\ E_1\ :=\ E_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$(\lambda l \in \mathsf{LV}.\ \lambda v \in \mathsf{RV}.\{\!|\ \mathsf{store}\ l\ v\ s\ |\!\})(\mathcal{E}[\![E_1]\!]e\ s)(\mathcal{R}[\![E_2]\!]e\ s)$$

$$\mathcal{S}[\![\ S_1\ ;\ S_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{fix}(\lambda f \in \mathsf{R} \rightarrow \mathsf{R}.\ \lambda r \in \mathsf{R}.$$
$$\mathsf{ext}[\mathcal{S}[\![S_2]\!]e,$$
$$\lambda(r' \in \mathsf{R}_\perp, s' \in \mathsf{S}).\ (f(r'), s')] \circ r)$$
$$(\mathcal{S}[\![S_1]\!]e)$$

$$\mathcal{S}[\![\ \mathbf{skip}\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{strict}\lambda s \in \mathsf{S}.\ \{\!|s|\!\}$$

$$\mathcal{S}[\![\ S_1\ \|\ S_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{fix}(\lambda g \in (\mathsf{R} \times \mathsf{R}) \rightarrow \mathsf{R}.$$
$$\lambda(r_1 \in \mathsf{R}, r_2 \in \mathsf{R}).\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$(\mathsf{ext}[r_2, \lambda(r_1' \in \mathsf{R}_\perp, s' \in \mathsf{S}).\ (g(r_1', r_2), s')](r_1(s)))\uplus$$
$$(\mathsf{ext}[r_1, \lambda(r_2' \in \mathsf{R}_\perp, s' \in \mathsf{S}).\ (g(r_1, r_2'), s')](r_2(s))))$$
$$(\mathcal{S}[\![S_1]\!]e)(\mathcal{S}[\![S_1]\!]e)$$

$$\mathcal{S}[\![\ <\ S_1\ >\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{fix}(\lambda h \in \mathsf{R} \rightarrow \mathsf{R}.\ \lambda r \in \mathsf{R}.$$
$$\mathsf{ext}[\lambda s \in \mathsf{S}.\ \{\!|s|\!\},$$
$$\lambda(r' \in \mathsf{R}, s' \in \mathsf{S}).\ h(r')(s')] \circ r)$$
$$(\mathcal{S}[\![S_1]\!]e)$$

Table 2.38: Denotations for interleaved statements

2.6 Bibliographical Notes

This final section refers to some published works on Denotational Semantics and related topics, and indicates their significance.

2.6.1 Development

The development of Denotational Semantics began with the paper "Towards a Formal Semantics" [52], written by Christopher Strachey in 1964 for the IFIP Working Conference on *Formal Language Description Languages*. The paper introduces compositionally-defined semantic functions that map abstract syntax to "operators" (i.e., functions), and it makes use of the fixed-point combinator, Y, for expressing the denotations of loops. It also introduces (compound) L-values and R-values, in connection with the semantics of assignment and parameter-passing. The treatment of identifier bindings follows Landin's approach [21]: identifiers are mapped to bound variables of λ-abstractions.

Strachey's paper "Fundamental Concepts of Programming Languages" [53] provides much of the conceptual analysis of programming languages that underlies their denotational semantics.

The main theoretical problem with Strachey's early work was that, formally, denotations were specified using the type-free λ-calculus, for which there was no known model. In fact Strachey was merely using λ-abstractions as a convenient way of expressing functions, rather than as a formal calculus. However, the fixed-point combinator Y was needed (for obtaining a compositional semantics for iterative constructs, for instance). Because Y involves self-application, it was considered to be "paradoxical": it could be interpreted operationally, but it could not be regarded as expressing a function. By 1969, Dana Scott had become interested in Strachey's ideas. In an exciting collaboration with Strachey, Scott first convinced Strachey to give up the type-free λ-calculus; then he discovered that it did have a model, after all. Soon after that, Scott established the Theory of Semantic Domains, providing adequate foundations for the semantic descriptions that Strachey had been writing.

The original paper on semantic domains by Scott [46] takes domains to be complete lattices (rather than the cpos used nowadays). Domains have effectively-given bases; Cartesian product, (coalesced) sum, and continuous function space are allowed as domain constructors; and solutions of domain equations are found as limits of sequences of embeddings. A domain providing a model for self-application (and hence for the λ-calculus) is given, and a recursively-defined domain for the denotations of storable procedures is proposed. (For references to subsequent presentations of domain theory, see [18].)

In a joint paper [48], Scott and Strachey present what is essentially the approach now known as Denotational Semantics (it was called "Mathematical Semantics" until 1976). The paper establishes meta-notation for defining semantic functions, and uses functional notation—rather than the λ-calculus—for specifying denotations. Here, for the first time, denotations are taken to be functions of environments, following a suggestion of Scott. The abstract syntax of finite programs is a set of derivation trees, although it is pointed out that this set could be made into a domain: then semantic functions are continuous, and their existence is guaranteed by the fixed point theorem

(see also [47], where partial and infinite programs are considered).

The notion of "characteristic domains" was introduced by Strachey in [54], where characteristic domains are given for ALGOL60 and for a pedagogical language (PAL).

The use of continuations in denotational semantics was proposed by Christopher Wadsworth, and reported in a joint paper with Strachey [55]. The present author was one of the first to exploit the technique, in a denotational description of ALGOL60 [28].

By the mid-1970's, sufficient techniques had been developed for specifying the denotational semantics of any conventional (sequential) programming language. Moreover, John Reynolds [44] and Robert Milne [22] had devised a way of proving the equivalence of denotational descriptions that involve different domains (e.g., direct and continuation semantics for the same language). Wadsworth had shown the relation between the computational and denotational semantics of the λ-calculus [59] (see also [36]). The present author had constructed a prototype "semantics implementation system" (SIS), for generating implementations of programming languages directly from their denotational descriptions [29, 30, 31]. Strachey's inspiration was sorely missed after his untimely death in 1975; but there was confidence that denotational semantics was the best approach to programming language semantics, and that it would be a routine matter to apply it to any real programming language.

Then the increasing interest in *concurrent* systems of processes led to the development of programming languages with non-deterministic constructs. An early treatment by Robin Milner [25] introduced a technique using so-called "oracles", but did not give sufficiently abstract denotations: for instance, non-deterministic choice was not commutative. Then Gordon Plotkin showed how to define power domains [40]. The introduction of power domains required domains to be cpos, rather than complete lattices. Moreover, for domains to be closed under power domain constructions, the cpos had to be restricted to be so-called SFP objects: limits of sequences of finite cpos (equivalent to the bifinite cpo's, see [18]). Much of Plotkin's paper is devoted to establishing the SFP framework. Also, the technique of "resumptions" is introduced, and used to define the denotations for some simple parallel programs.

Mike Smyth gave a simple presentation of Plotkin's power domains [49] (and introduced a "weak" power domain). Matthew Hennessy and Plotkin together defined a category of "non-deterministic" domains [20], and showed that the (Plotkin) power domain, $D^\natural$, of a domain D is just the free continuous semi-lattice generated by D. They also introduced a tensor product for non-deterministic domains, and obtained full abstractness for a simple (although somewhat artificial) parallel programming language. Krzysztof Apt and Plotkin [3] related the Plotkin power domain to operational semantics; they showed that Smyth's weak power domain (of states) corresponds to Dijkstra's predicate transformers. Plotkin [41] generalized power domains to deal with countable non-determinism. Samson Abramsky has shown [1] that the Plotkin power domain gives fully abstract denotations when observable behaviour is characterized by classes of finite experiments. There has also been work on power domains using complete metric spaces [2].

Despite all the above works, it is debatable whether the denotational treatment of concurrency is satisfactory. There are difficulties with getting reasonable abstractness of denotations when

using resumptions. Moreover, the use of power domains gives an unwelcome notational burden. In contrast, Structural Operational Semantics (illustrated in [24]) extends easily from sequential languages to concurrency.

Another problem with the applicability of Denotational Semantics concerns the *pragmatic* aspects of denotational descriptions. For "toy" languages, it is quite a simple matter to "lay the domains on the table" (following [54]), and to give semantic equations that define appropriate (but not necessarily fully abstract) denotations. However, the approach does not scale up easily to "real" programming languages, which (unfortunately) seem to require a large number of complex domains for their denotational semantics. Partly because the semantic equations depend explicitly on the domains of denotations, it can be extremely difficult to comprehend a large denotational description.

A related problem is that it is not feasible to re-use parts of the description of one language (PASCAL, say) in the description of another language (MODULA2, for instance). Analogous problems in software engineering were alleviated by the introduction of "modules". Denotational Semantics has no notation for expressing modules. In fact if the definitions of the domains of denotations were to be encapsulated in modules, it would not be possible to express denotations using λ-notation in the semantic equations: one would have to use auxiliary operations, defined in the modules, for expressing primitive denotations and for combining denotations. Thus it seems that a high degree of modularity is incompatible with (conventional) denotational semantics.

An aggravating factor, concerning the problem of (writing and reading) large denotational descriptions, may be that the intimate relation between higher-order functions on domains and computational properties is not immediately apparent. (For example, with non-strict functions, arguments may not need to be evaluated.) It is difficult for the non-specialist to appreciate the abstract denotations of programming constructs.

The effort required to formulate a denotational semantics for a real programming language is reflected by the lack of published denotational descriptions of complete, real programming languages. Efforts have been made for SNOBOL [56], ALGOL60 [28], ALGOL68 [22], PASCAL [58], and ADA [11]. In general, these descriptions make some simplifying assumptions about the programming language concerned; they also omit the definitions of various "primitive" functions, and use numerous notational conventions whose formal status is somewhat unclear. (Of course, much the same—and other—criticisms could be made of alternative forms of semantics.)

Hope for the future of denotational semantics lies in the recent popularization of two languages that have been designed with formal (denotational) semantics in mind: Standard ML [19], and Scheme [43]. Although the denotational descriptions of these languages are not used formally as standards for implementations, they do show that it is possible to give complete descriptions of useful languages.

2.6.2 Exposition

There are several expository works that explain the basic notions of Denotational Semantics, and give examples of the prevailing techniques for choosing denotations:

Bob Tennent [57] provides a basic tutorial introduction, containing a semantic description of Reynold's experimental language GEDANKEN and a useful bibliography.

The epic work by Milne and Strachey [23], completed by Milne after Strachey's death, contains careful discussions of many techniques for choosing denotations, including less abstract "non-standard" denotations. The examples given are related to ALGOL68. It is a valuable reference for further study of Denotational Semantics.

Joe Stoy's book [50] is partly based on Strachey's lectures at Oxford; consequently, scant attention is paid to the syntactic constructs of later programming languages, such as PASCAL. The foreword by Scott gives an detailed appreciation of Strachey and his work.

The book by Mike Gordon [16] takes an engineering approach: it does not explain foundations at all. The techniques illustrated are adequate for the description of most Pascal-like programming languages.

The introductory book on Denotational Semantics by Dave Schmidt [45] includes a rather comprehensive description of domain theory (including power domains). The book covers a number of incidental topics, such as semantics-directed compiler generation, and there is a substantial bibliography.

Unfortunately, there is considerable variation in the notation (and notational conventions) used in the works referenced above: almost the only common notational feature is the use of 'λ' for function abstraction and juxtaposition for function application! The reader should be prepared to adapt not only to different symbols used for the same constants and operators, but also to different choices of what to regard as primitive and what to define as auxiliary notation. (N.B. the notation presented and used in this chapter is *not* an accepted standard.)

2.6.3 Variations

The approach to semantics presented in this chapter, whose development is sketched above, may be regarded as the main theme of Denotational Semantics: abstract syntax, domains of denotations, semantic functions defined by semantic equations using λ-notation. Some significant variations on this theme are indicated below.

Initial Algebra Semantics

(This approach was sketched in Sections 2.2 and 2.3.) Initial Algebra Semantics was developed by Joseph Goguen, Jim Thatcher, Eric Wagner, and Jesse Wright [15]. Although it is formally equivalent to denotational semantics, it has the advantage of making it explicit that abstract syntax is an initial algebra, and that semantic functions are homomorphic. Explicit structural induction proofs in denotational semantics can here be replaced by appeals to initiality. It is easy to extend Initial Algebra Semantics to continuous algebras, so as to allow infinite programs, whereas

with denotational semantics, abstract syntax has to be changed from sets to cpos. An additional benefit of Initial Algebra Semantics is that one always names the domains of denotations; this seems to encourage the specification of denotations as *compositions*, rather than as applications and abstractions (but this is only a matter of style). Initial Algebra Semantics has not yet been applied to real programming languages.

The French school of Algebraic Semantics [17] has concentrated on the semantics of program schemes, rather than of particular programming languages. (See [9].)

OBJ

Goguen and Kamran Parsaye-Ghomi show in [14] how the algebraic specification language OBJT (a precursor of OBJ2 [13]) can be used to give modular semantic descriptions of programming languages. Their framework is first-order, and not strictly compositional; but higher-order algebras, which give the power of λ-notation in an algebraic framework [39, 42, 12], could be used instead of OBJ, with similar modularity.

Despite the use of explicit modules, the semantic equations given by Goguen and Parsaye-Ghomi are still sensitive to the functionality of denotations. The approach has not been applied to real programming languages.

VDM

The Vienna Development Method, VDM [5], has an elaborate notation, called META-IV, that can be used to give denotational descriptions of programming languages. Although there are quite a few variants of META-IV, these share a substantial, partly-standardized auxiliary notation that provides a number of useful "flat" domain constructors (e.g., sets, maps) and declarative and imperative constructs (e.g., let-constructions, storage allocation, sequencing, exception-handling). However, this auxiliary notation is a supplement to, rather than a replacement for, the λ-notation. The foundations of META-IV have been investigated by Stoy [51] and Brian Monahan [27].

In contrast to Denotational Semantics, VDM avoids the use of high-order functions and non-strict abstractions, in order to keep close to the familiar objects of conventional programming. (Andrzej Blikle and Andrzej Tarlecki [7] went even further, and advocated avoidance of reflexive domains.) But as in Denotational Semantics, there are severe problems with large-scale descriptions, due to the lack of modularity. The fact that it has been possible to develop semantic descriptions of real programming languages such as CHILL [8] and ADA [6] in (extended versions of) META-IV is a tribute to the discipline and energy of their authors, rather than evidence of an inherent superiority of META-IV over Denotational Semantics.

Action Semantics

Action Semantics [38, 37, 33, 60, 35, 32] is something of a mixture of the denotational, algebraic, and operational approaches to formal semantics. It has been under development since 1977, by

the present author and (since 1984) David Watt. A brief summary of the main features of Action Semantics is given below.

The primary aim is to make it easier to deal with semantic descriptions of "real" programming languages. Factors that have been addressed include modularity (to obtain ease of modification, extension, and re-use) and notation (to improve comprehensibility).

Action Semantics is compositional, just like Denotational Semantics. The essential difference between Action Semantics and Denotational Semantics concerns the entities that are taken as the denotations of program phrases: so-called "actions", rather than functions on semantic domains. Actually, some actions do correspond closely to functions, and are determined purely by the relation between the "information" that they receive and produce. But other actions have a more operational essence: they process information *gradually*, and they may interfere (or collaborate) when put together.

The standard notation for actions is polymorphic, in that actions may be combined without regard to what "kind" of information they process: transient or stored data, bindings, or communications. Furthermore, the different kinds of information are processed independently. This allows the semantic equations for (say) arithmetical expressions to stay the same, even when expressions might be polluted with "side-effects" or communications.

The fundamental concepts of programming languages (as identified by Strachey, and implicit in most denotational descriptions) can be expressed straightforwardly in action notation. The comprehensibility of action semantic descriptions is enhanced by the use of suggestive words, rather than (to programmers) cryptic mathematical symbols: in the usual concrete representation of action notation, for example, the action combinator expressing sequential performance is written as infixed 'then'. The notation is claimed to be a reasonable "compromise" between previous formal notations and informal English.

The theory of actions includes some pleasant algebraic laws. However, the basic understanding of actions is operational, and action equivalence is defined by a (structural) operational semantics [32]. Action notation may also be defined denotationally, and used as auxiliary notation in conventional denotational descriptions, as illustrated in [34].

At the time of writing, it is not apparent whether Action Semantics will turn out to be any more palatable than Denotational Semantics to the programming community.

Bibliography

[1] S. Abramsky. Experiments, powerdomains, and fully abstract models for applicative multiprogramming. In *FCT'83, Proc. Int. Conf. on Foundations of Computation Theory*, number 158 in Lecture Notes in Computer Science, pages 1–13. Springer-Verlag, 1983.

[2] P. America, J. de Bakker, J. N. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 1989. To appear.

[3] K. R. Apt and G. D. Plotkin. A cook's tour of countable nondeterminism. In *ICALP'81, Proc. Int. Coll. on Automata, Languages, and Programming, Haifa*, number 115 in Lecture Notes in Computer Science, pages 479–494. Springer-Verlag, 1981.

[4] H. Barendregt. Functional programming and lambda calculus. This handbook.

[5] D. Bjørner and C. B. Jones, editors. *Formal Specification & Software Development.* Prentice-Hall, 1982.

[6] D. Bjørner and O. N. Oest, editors. *Towards a Formal Description of Ada.* Number 98 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[7] A. Blikle and A. Tarlecki. Naive denotational semantics. In *Information Processing 83, Proc. IFIP Congress 83.* North-Holland, 1983.

[8] CCITT. *CHILL Language Definition, Recommendation Z200*, May 1980.

[9] B. Courcelle. Recursive applicative program schemes. This handbook.

[10] P. Cousot. Hoare logic. This handbook.

[11] V. Donzeau-Gouge, G. Kahn, B. Lang, et al. *Formal Definition of the Ada Programming Language, Preliminary Version.* INRIA, 1980.

[12] P. Dybjer. Domain algebras. In *ICALP'84, Proc. Int. Coll. on Automata, Languages, and Programming, Antwerp*, number 172 in Lecture Notes in Computer Science. Springer-Verlag, 1984.

[13] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *POPL'85, Proc. 12th Ann. ACM Symp. on Principles of Programming Languages.* ACM, 1985.

[14] J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In *Proc. Int. Coll. on Formalization of Programming Concepts, Peñiscola*, number 107 in Lecture Notes in Computer Science. Springer-Verlag, 1981.

[15] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24:68–95, 1977.

[16] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[17] I. Guessarian. *Algebraic Semantics.* Number 99 in Lecture Notes in Computer Science. Springer-Verlag, 1981.

[18] C. A. Gunter and D. S. Scott. Semantic domains. This handbook.

[19] R. Harper, D. MacQueen, and R. Milner. Standard ML. Report ECS-LFCS-86-2, Computer Science Dept., University of Edinburgh, 1986.

[20] M. C. B. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In *MFCS'79, Proc. Symp. on Math. Foundations of Computer Science*, number 74 in Lecture Notes in Computer Science. Springer-Verlag, 1979.

[21] P. J. Landin. A formal description of Algol60. In *Formal Language Description Languages for Computer Programming, Proc. IFIP TC2 Working Conference, 1964*, pages 266–294. IFIP, North-Holland, 1966.

[22] R. E. Milne. *The Formal Semantics of Computer Languages and Their Implementations.* PhD thesis, University of Cambridge, 1974. Available as Tech. Microfiche TCF–2, Programming Research Group, University of Oxford, 1974.

[23] R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman & Hall, 1976.

[24] R. Milner. Operational and algebraic semantics of concurrent processes. This handbook.

[25] R. Milner. An approach to the semantics of parallel programs. In *Proc. Convegno di Informatica Teoretica.* Instituto di Elaborazione della Informazione, Pisa, 1973.

[26] J. C. Mitchell. Types in programming. This handbook.

[27] B. Q. Monahan. A type model for VDM. In D. Bjørner, C. B. Jones, et al., editors, *VDM'87, VDM – A Formal Method at Work*, number 252 in Lecture Notes in Computer Science, pages 210–236. Springer-Verlag, 1987.

[28] P. D. Mosses. The mathematical semantics of Algol60. Tech. Mono. PRG–12, Programming Research Group, University of Oxford, 1974.

[29] P. D. Mosses. *Mathematical Semantics and Compiler Generation.* D.Phil. dissertation, University of Oxford, 1975.

[30] P. D. Mosses. Compiler generation using denotational semantics. In *MFCS'76, Proc. Symp. on Math. Foundations of Computer Science, Gdańsk*, number 45 in Lecture Notes in Computer Science. Springer-Verlag, 1976.

[31] P. D. Mosses. SIS, Semantics Implementation System: Reference manual and user guide. Tech. Mono. MD–30, Computer Science Dept., Aarhus University, 1979. Note: the system SIS itself is no longer available.

[32] P. D. Mosses. Action Semantics. Draft, Version 6, 1988.

[33] P. D. Mosses. The modularity of action semantics. Internal Report DAIMI IR–75, Computer Science Dept., Aarhus University, 1988.

[34] P. D. Mosses. A practical introduction to denotational semantics. Draft, 1989.

[35] P. D. Mosses. Unified algebras and action semantics. In *STACS'89, Proc. Symp. on Theoretical Aspects of Computer Science*, number ??? in Lecture Notes in Computer Science. Springer-Verlag, 1989.

[36] P. D. Mosses and G. D. Plotkin. On limiting completeness. *SIAM J. Comput.*, 16:179–194, 1987.

[37] P. D. Mosses and D. A. Watt. Pascal: Action Semantics. Draft, Version 0.3, August 1986.

[38] P. D. Mosses and D. A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986.* IFIP, North-Holland, 1987.

[39] K. Parsaye-Ghomi. *Higher Order Data Types.* PhD thesis, Computer Science Dept., UCLA, 1981.

[40] G. D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487, 1976.

[41] G. D. Plotkin. A powerdomain for countable non-determinism. In *ICALP'82, Proc. Int. Coll. on Automata, Languages, and Programming, Aarhus*, number 140 in Lecture Notes in Computer Science, pages 418–428. Springer-Verlag, 1982.

[42] A. Poigné. On semantic algebras. Tech. report, Informatik II, Universität Dortmund, 1983.

[43] J. Rees, W. Clinger, et al. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.

[44] J. C. Reynolds. On the relation between direct and continuation semantics. In *ICALP'74, Proc. Int. Coll. on Automata, Languages, and Programming, Saarbrücken*, number 14 in Lecture Notes in Computer Science, pages 157–168. Springer-Verlag, 1974.

[45] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn & Bacon, 1986.

[46] D. S. Scott. Outline of a mathematical theory of computation. In *Proc. Fourth Annual Princeton Conference on Information Sciences and Systems*, 1970. A revised and slightly expanded version is Tech. Mono. PRG–2, Programming Research Group, University of Oxford, 1970.

[47] D. S. Scott. The lattice of flow diagrams. In *Symposium on Semantics of Algorithmic Languages*, number 188 in Lecture Notes in Mathematics. Springer-Verlag, 1971.

[48] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series.* Polytechnic Institute of Brooklyn, 1971.

[49] M. B. Smyth. Power domains. *J. Comput. Syst. Sci.*, 16:23–36, 1978.

[50] J. E. Stoy. *The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[51] J. E. Stoy. Mathematical foundations. In Bjørner and Jones [5], chapter 3.

[52] C. Strachey. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming, Proc. IFIP TC2 Working Conference, 1964*, pages 198–220. IFIP, North-Holland, 1966.

[53] C. Strachey. Fundamental concepts of programming languages. Lecture Notes for a NATO Summer School, Copenhagen. Available from Programming Research Group, University of Oxford, 1967.

[54] C. Strachey. The varieties of programming language. In *Proc. International Computing Symposium*, pages 222–233. Cini Foundation, Venice, 1972. A revised and slightly expanded version is Tech. Mono. PRG–10, Programming Research Group, University of Oxford, 1973.

[55] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Tech. Mono. PRG–11, Programming Research Group, University of Oxford, 1974.

[56] R. D. Tennent. Mathematical semantics of SNOBOL4. In *POPL'73, Proc. Ann. ACM Symp. on Principles of Programming Languages*, pages 95–107. ACM, 1973.

[57] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19:437–453, 1976.

[58] R. D. Tennent. A denotational description of the programming language Pascal. Technical report, Programming Research Group, University of Oxford, 1978.

[59] C. P. Wadsworth. The relation between the computational and denotational properties for Scott's D_∞-models of the lambda-calculus. *SIAM J. Comput.*, 5:488–521, 1976.

[60] D. A. Watt. An action semantics of Standard ML. In *Proc. Fourth Workshop on Math. Foundations of Programming Language Semantics, Boulder*, number 298 in Lecture Notes in Computer Science, pages 572–598. Springer-Verlag, 1988.