ISSN 0105-8517

Semantic Domains and Denotational Semantics

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

DAIMI PB – 276
April 1989

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.

Chapter 1

Semantic Domains.

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.**

List of Figures

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

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, $\bot, \mathsf{true}, \mathsf{false}$, where $\bot \sqsubseteq \mathsf{true}$ and $\bot \sqsubseteq \mathsf{false}$ (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 $f : D \circ\!\!\to E$ 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 $D \circ\!\!\to E$ 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 $D \circ\!\!\to E$.)

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.

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

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

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 $f : \mathsf{N}_\perp \circ\!\!\to \mathsf{N}_\perp$. 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.

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 $h : D \circ\!\!\to E$ which makes the following diagram commute

$$\begin{tikzcd}
D \arrow[r, "f"] \arrow[d, "h"'] & D \arrow[d, "h"] \\
E \arrow[r, "g"] & E
\end{tikzcd}$$

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

$$\begin{tikzcd}
D' \arrow[r, "f'"] \arrow[d, "i"'] & D' \arrow[d, "i"] \\
D \arrow[r, "f"] & D
\end{tikzcd}$$

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.

1.3.1 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 $\bot \in A$.*
4. *$\langle \mathcal{P}(C), \lhd \rangle$ is a cpo with $\{\bot\}$ 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 $\mathsf{strict} : (D \to E) \to (D \circ\!\!\to E)$ given by:

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

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

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 $\mathcal{P}\mathsf{N}$ is an example of an algebraic lattice. On the other hand, the bounded complete domain $\mathsf{N}_{\perp} \circ\!\!\to \mathsf{N}_{\perp}$ 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 $D \circ\!\!\to E$.*

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 $\mathsf{step}(s) : D \to E$ given by taking $\mathsf{step}(s)(x) = \bigsqcup\{f(y) \mid y \in N \cap \downarrow x\}$ 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 $D \circ\!\!\to E$ 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 $\cdot \circ\!\!\to \cdot$. 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.

$$\begin{tikzcd}
D & \\
D \times E \arrow[u, "\mathsf{fst}"] \arrow[d, "\mathsf{snd}"'] & F \arrow[ul, "f"'] \arrow[l, dashed, "\langle f, g \rangle"'] \arrow[dl, "g"] \\
E &
\end{tikzcd}$$

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:

$$\begin{tikzcd}
D \times E \arrow[r, "f"] \arrow[d, dashed, "\mathsf{curry}(f) \times \mathsf{id}"'] & F \\
(E \to F) \times E \arrow[ur, "\mathsf{apply}"'] &
\end{tikzcd}$$

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 $h : \mathsf{N} \rightarrow \mathsf{N}$ 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:

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

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

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

given by taking

$$\mathsf{smash}(x, y) = \begin{cases} (x, y) & x \neq \bot \text{ and } x \neq \bot \\ \bot_{D\otimes E} & \text{otherwise} \end{cases}$$

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 $\mathsf{unsmash} : D \otimes E \to D \times E$ given by

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

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

$$\begin{tikzcd}
D \times E \arrow[d, "\mathsf{smash}"'] \arrow[dr, "f"] & \\
D \otimes E \arrow[r, dashed, "g"] & F
\end{tikzcd}$$

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

$$\begin{tikzcd}
D \times E \arrow[r, "f \times g"] \arrow[d, "\mathsf{smash}"'] & D \times E \arrow[d, "\mathsf{smash}"] \\
D \otimes E \arrow[r, dashed, "f \otimes g"] & D \otimes E
\end{tikzcd}$$

As with the product $\times$ and function space $\to$, there is a relationship between the smash product $\otimes$ and the strict function space $\circ\!\!\to$. In particular, there is a strict continuous function $\mathsf{strict\_apply}$ such that for any strict function f, there is a unique strict function $\mathsf{strict\_curry}$ such that the following diagram commutes:

$$\begin{tikzcd}
D \otimes E \arrow[r, "f"] \arrow[d, dashed, "\mathsf{strict\_curry}(f) \otimes \mathsf{id}"'] & F \\
(E \circ\!\!\to F) \otimes E \arrow[ur, "\mathsf{strict\_apply}"'] &
\end{tikzcd}$$

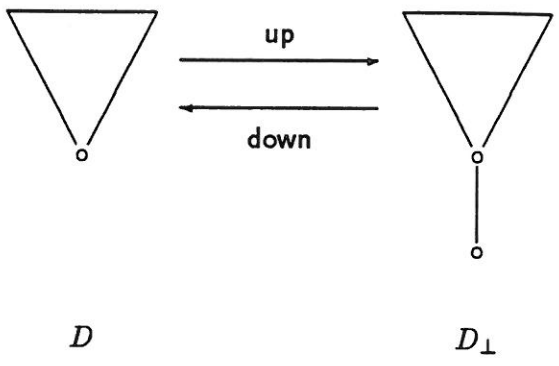

Figure 1.2: The lift of a cpo.

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:

$$\begin{tikzcd} D \arrow[d, "\mathsf{up}"'] \arrow[dr, "f"] & \\ D_\perp \arrow[r, dashed, "f^\dagger"] & E \end{tikzcd}$$

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

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

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].

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

$u \subseteq x$ 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 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:*

$$\begin{tikzcd}
D \arrow[d, "\{\!|\cdot|\!\}"'] \arrow[dr, "f"] & \\
D^\natural \arrow[r, dashed, "\mathsf{ext}(f)"] & E
\end{tikzcd}$$

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 $\mathcal{P}_f^*(\mathsf{T} \times \mathsf{T})$:

$$\begin{aligned} u &= \{\langle \bot, \mathsf{true}\rangle,\ \langle \bot, \mathsf{false}\rangle\} \\ v &= \{\langle \mathsf{true}, \bot\rangle,\ \langle \mathsf{false}, \bot\rangle\} \\ u' &= \{\langle \mathsf{true}, \mathsf{true}\rangle,\ \langle \mathsf{false}, \mathsf{false}\rangle\} \\ v' &= \{\langle \mathsf{true}, \mathsf{false}\rangle,\ \langle \mathsf{false}, \mathsf{true}\rangle\} \end{aligned}$$

It is not hard to see that u' and v' are *minimal* upper bounds for $\{u, v\}$ with respect to the ordering $\vdash^\natural$. 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 $N \triangleleft A$ 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 $N \triangleleft A$ with $u \subseteq N$. ▮

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

1. *D 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 $N \lhd K(D)$. 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 $S_f \subseteq N_f \lhd K(D)$. 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 $D \to E$, $D \circ\!\!\to E$, $D \times E$, $D \otimes E$, $D + E$, $D \oplus E$, $D_\perp$, $D^\natural$, $D^\sharp$ 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:

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):

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

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

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 *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:

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

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):

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: $\to$, $\circ\!\!\to$, $\times$, $\otimes$, $+$, $\oplus$, $(\cdot)_\perp$, $(\cdot)^\sharp$, $(\cdot)^\flat$. ▮

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:

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.

[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.

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-

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 $(a, (x_1 \dots x_n))$ as $a ::= x_1 \dots x_n$. 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.

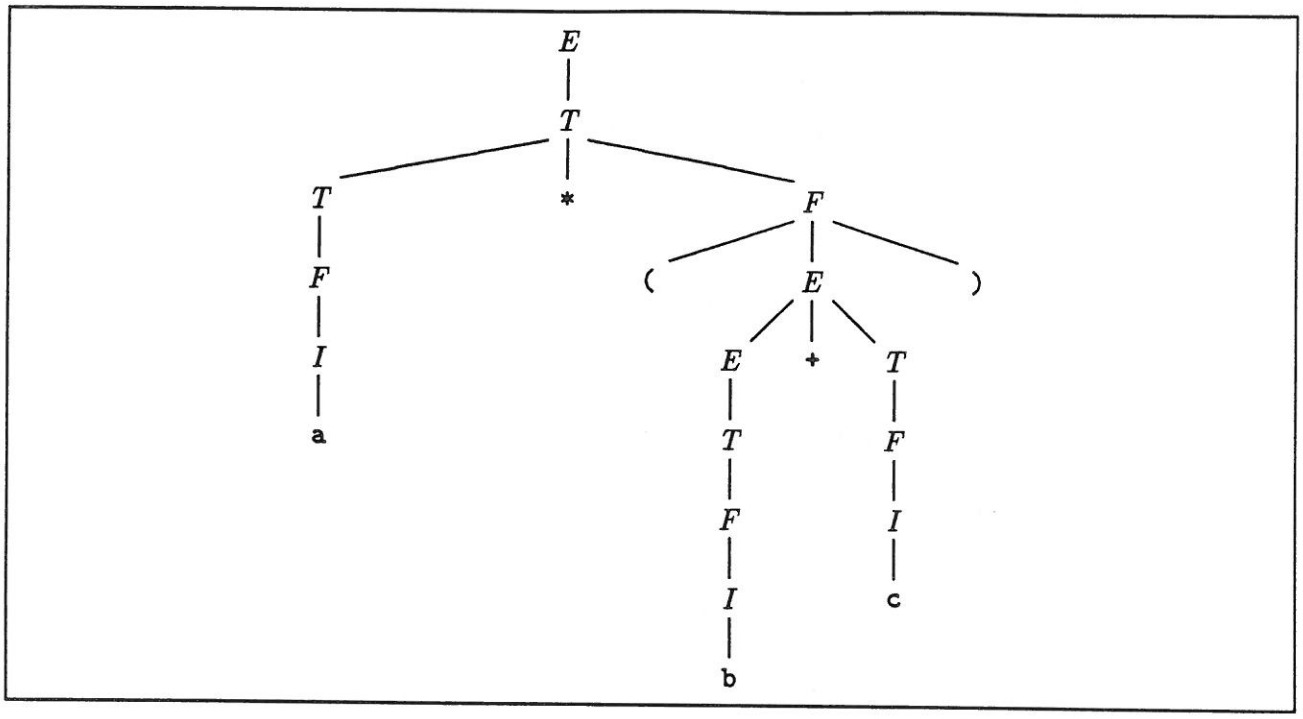

Figure 2.1: A derivation tree for concrete syntax

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

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

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.

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 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 N, and such 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

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

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.

$$\mathcal{Z} : \textsc{Signed-Binary-Numeral} \to \mathsf{Z}$$

$$\begin{aligned}
\mathcal{Z}[\![\ B\ ]\!] &= \mathcal{B}[\![B]\!] \\
\mathcal{Z}[\![\ \texttt{-}\ B\ ]\!] &= -\mathcal{B}[\![B]\!]
\end{aligned}$$

$$\mathcal{B} : \textsc{Binary-Numeral} \to \mathsf{N}$$

$$\begin{aligned}
\mathcal{B}[\![\ \texttt{0}\ ]\!] &= 0 \\
\mathcal{B}[\![\ \texttt{1}\ ]\!] &= 1 \\
\mathcal{B}[\![\ B\ \texttt{0}\ ]\!] &= 2 \times (\mathcal{B}[\![B]\!]) \\
\mathcal{B}[\![\ B\ \texttt{1}\ ]\!] &= (2 \times (\mathcal{B}[\![B]\!])) + 1
\end{aligned}$$

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*.

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 $\mathsf{graph}(f) \subseteq \mathsf{graph}(g)$. 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.

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)

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

(Domain-Variables)

w ::= *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 ::= *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

$$D = D \rightarrow D$$

Function Domains

$d_1 \rightarrow d_2$ 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 $f \sqsubseteq_{d_1 \rightarrow d_2} g$ 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 $f : d \rightarrow d'$ 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 $f_1 : d' \rightarrow d''$ and $f_2 : d \rightarrow d'$ 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 $d \rightarrow d$ to the least solution x of the equation $x = f(x)$.

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

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

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 $\perp$.

$e_1 \uplus e_2$ 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 $d_1{}^\natural \rightarrow d_2{}^\natural$, where e denotes a function in $d_1 \rightarrow d_2$.

ext_d extends functions in $d = d_1 \rightarrow d_2{}^\natural$ to $d_1{}^\natural \rightarrow d_2{}^\natural$.

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 $\rightarrow$ and $\circ\!\!\rightarrow$ associate to the right, and have weaker precedence than $+$, $\oplus$, $\times$, and $\otimes$:
  $D_1 \times D_2 \rightarrow D_3 \rightarrow D_4$ is grouped as $(D_1 \times D_2) \rightarrow (D_3 \rightarrow D_4)$.
- 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 $\mathsf{in}_1(\perp_\mathsf{O})$;
  - $d^* \cong (\mathsf{O} \oplus (d \otimes d^*))$;
  - $d^\infty \cong (d \times d^\infty)$;

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:

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

Table 2.9: Domains for literals

$$\mathcal{L} : \textsc{Literal} \rightarrow \mathsf{V}$$

$$\mathcal{N} : \textsc{Numeral} \rightarrow \mathsf{Num}$$

$$\mathcal{C} : \textsc{Character} \rightarrow \mathsf{Char}$$

$$\mathcal{CS} : \textsc{Character-String} \rightarrow \mathsf{String}$$

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, $\sqsubseteq$:

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

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

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

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

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

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

$\mathcal{MO} : \textsc{Monadic-Operator} \to (\mathsf{V} \circ\!\!\to \mathsf{V})$

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

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

$\mathcal{DO} : \textsc{Dyadic-Operator} \to (\mathsf{V} \otimes \mathsf{V} \circ\!\!\to \mathsf{V})$

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

...

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

...

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

Table 2.12: Denotations for expressions

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

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 $\mathcal{CD}[\![$ `rec val` I `=` E $]\!](e)$. From the semantic equations we have

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

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., 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 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 PV = FV = F, leaving essentially F = F $\rightarrow$ 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',*

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

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'.

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 | VD₁, VD₂
(TYPE)
    T  ::= bool | num | T[1..N]
(EXPRESSION)
    E  ::= E₁[E₂]
```

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

S	$=$	$\mathsf{Loc} \to (\mathsf{SV} \oplus \mathsf{T})$
Loc	$=$	$0 \oplus \mathsf{Loc}$
empty	$=$	$\lambda l \in \mathsf{Loc}.\ \mathsf{false}$
	$\in$	S
$\mathsf{reservation}$	$=$	$\lambda l \in \mathsf{Loc}.\ \lambda s \in \mathsf{S}.$ $(\lambda l' \in \mathsf{Loc}.\ \mathsf{if}\ l =_{\mathsf{Loc}} l'\ \mathsf{then\ true\ else}\ s(l'))$
	$\in$	$\mathsf{Loc} \to \mathsf{S} \to \mathsf{S}$
$\mathsf{freedom}$	$=$	$\lambda l \in \mathsf{Loc}.\ \lambda s \in \mathsf{S}.$ $(\lambda l' \in \mathsf{Loc}.\ \mathsf{if}\ l =_{\mathsf{Loc}} l'\ \mathsf{then\ false\ else}\ s(l'))$
	$\in$	$\mathsf{Loc} \to \mathsf{S} \to \mathsf{S}$
store	$=$	$\lambda l \in \mathsf{Loc}.\ \lambda v \in \mathsf{SV}.\ \lambda s \in \mathsf{S}.$ $(\lambda l' \in \mathsf{Loc}.\ \mathsf{if}\ l =_{\mathsf{Loc}} l'\ \mathsf{then}\ v\ \mathsf{else}\ s(l'))$
	$\in$	$\mathsf{Loc} \to \mathsf{SV} \to \mathsf{S} \to \mathsf{S}$
stored	$=$	$\lambda l \in \mathsf{Loc}.\ \lambda s \in \mathsf{S}.\ [\mathsf{id}_{\mathsf{SV}}, \perp](s(l))$
	$\in$	$\mathsf{Loc} \to \mathsf{S} \to \mathsf{SV}$
$\mathsf{location}$	$=$	*unspecified*
	$\in$	$\mathsf{S} \to \mathsf{Loc}$
$\mathsf{allocation}$	$=$	$\lambda s \in \mathsf{S}.\ (\lambda l \in \mathsf{Loc}.\ (l, \mathsf{reservation}\ l\ s))(\mathsf{location}\ s)$
	$\in$	$\mathsf{S} \to \mathsf{Loc} \times \mathsf{S}$

Table 2.21: Notation for stores

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

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

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

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

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

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

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

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

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

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

$\mathcal{T}[\![\ T\texttt{[1..}N\texttt{]}\ ]\!] = \mathsf{allocations}(\mathcal{T}[\![T]\!], \mathcal{N}[\![N]\!])$

$\mathcal{TU} : \textsc{Type} \rightarrow \mathsf{LV} \circ\!\!\rightarrow \mathsf{S} \circ\!\!\rightarrow \mathsf{S}$

$\mathcal{TU}[\![\ \texttt{bool}\ ]\!] = \mathsf{freedom}$

$\mathcal{TU}[\![\ \texttt{num}\ ]\!] = \mathsf{freedom}$

$\mathcal{TU}[\![\ T\texttt{[1..}N\texttt{]}\ ]\!] = \mathsf{freedoms}(\mathcal{TU}[\![T]\!], \mathcal{N}[\![N]\!])$

Table 2.23: Denotations for variable declarations and types

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

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

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

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

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

$\mathcal{R}$: $\textsc{Expression} \rightarrow \mathsf{Env} \rightarrow \mathsf{S} \rightarrow \mathsf{RV}$

$\mathcal{R}[\![\ E\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ [\mathsf{id}_{\mathsf{RV}}, \perp, \lambda l \in \mathsf{LV}.\ \mathsf{assigned}\ l\, s](\mathcal{E}[\![E]\!] e\, s)$

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

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

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

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

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

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

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

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

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

$\mathcal{E}[\![\ E_1\texttt{[}E_2\texttt{]}\ ]\!] = \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}.\ \mathsf{component}(\mathcal{E}[\![E_1]\!] e\, s, \mathcal{R}[\![E_2]\!] e\, s)$

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

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

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

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

Table 2.24: Denotations for expressions (modified)

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 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)
$E ::= \texttt{proc}\ (PD)\ S_1$

(PARAMETER-DECLARATION)
$PD ::= \texttt{var}\ I\texttt{:}\ T \mid I\texttt{:}\ T$

(STATEMENTS)
$S ::= E_1(E_2)$

Table 2.28: Syntax for procedures

$\mathsf{P} = (\mathsf{PV} \circ\!\!\rightarrow \mathsf{S} \circ\!\!\rightarrow \mathsf{S})_\perp$

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

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

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

$\mathcal{E}$: EXPRESSION $\rightarrow \mathsf{Env} \rightarrow \mathsf{S} \rightarrow \mathsf{EV}$

$\mathcal{E}[\![$ `proc (`PD`)` S $]\!] =$

$\quad \lambda e \in \mathsf{Env}.\ \mathsf{up}(\mathsf{strict}\lambda v \in \mathsf{PV}.$
$\qquad\qquad (\lambda(e' \in \mathsf{Env}, s \in \mathsf{S}).\ \mathcal{PU}[\![PD]\!]e'(\mathcal{S}[\![S]\!](\mathsf{overlay}(e', e))))$
$\qquad\qquad \circ\, \mathcal{PD}[\![PD]\!]e)$

$\mathcal{PD}$: PARAMETER-DECLARATION $\rightarrow \mathsf{PV} \rightarrow \mathsf{S} \rightarrow \mathsf{Env} \times \mathsf{S}$

$\mathcal{PD}[\![$ `val` I: T $]\!] = \lambda v \in \mathsf{PV}.\ \lambda s \in \mathsf{S}.\ (\mathsf{binding}\ I\, v, s)$

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

$\mathcal{PD}[\![$ I: T $]\!] = \lambda v \in \mathsf{PV}.\ \lambda s \in \mathsf{S}.$
$\qquad (\lambda v' \in \mathsf{RV}.\ (\lambda(l' \in \mathsf{LV}, s' \in \mathsf{S}).\ (\mathsf{binding}\ I\, l', \mathsf{assign}\ l'\, v'\, s'))$
$\qquad\qquad (\mathcal{T}[\![T]\!]s))$
$\qquad ([\lambda l \in \mathsf{LV}.\ \mathsf{assigned}\ l\, s, \mathsf{id}_{\mathsf{RV}}, \perp, \perp](v))$

$\mathcal{PU}$: PARAMETER-DECLARATION $\rightarrow \mathsf{Env} \rightarrow \mathsf{S} \rightarrow \mathsf{S}$

$\mathcal{PU}[\![$ `val` I $]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{id}_{\mathsf{S}}$

$\mathcal{PU}[\![$ `var` I: T $]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{id}_{\mathsf{S}}$

$\mathcal{PU}[\![$ I: T $]\!] = \lambda e \in \mathsf{Env}.\ \mathcal{TU}[\![T]\!](\mathsf{bound}\ I\, e)$

$\mathcal{S}$: STATEMENTS $\rightarrow \mathsf{Env} \rightarrow \mathsf{S} \circ\!\!\rightarrow \mathsf{S}$

$\mathcal{S}[\![$ 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

$$\mathsf{In} = \mathsf{SV}^*$$

$$\mathsf{Out} = \mathsf{SV}^*$$

$$\mathcal{P} : \textsc{Program} \to (\mathsf{In} \circ\!\!\to \mathsf{Out})$$

$$\mathcal{P}[\![\ \texttt{prog}\ S_1\ ]\!] = \lambda i \in \mathsf{In}.\ \mathsf{on}_3(\mathcal{S}[\![S_1]\!](\mathsf{void})(\mathsf{empty}, i, \top))$$

$$\mathcal{S} : \textsc{Statements} \to \mathsf{Env} \to (\mathsf{S} \otimes \mathsf{In} \otimes \mathsf{Out}) \circ\!\!\to (\mathsf{S} \otimes \mathsf{In} \otimes \mathsf{Out})$$

$$\begin{aligned}\mathcal{S}[\![\ \texttt{read}\ E\ ]\!] = {} & \lambda e \in \mathsf{Env}.\ \lambda(s \in \mathsf{S}, i \in \mathsf{In}, o \in \mathsf{Out}).\\ & (\lambda l \in \mathsf{Loc}.\ [\bot, \lambda(v \in \mathsf{SV}, i' \in \mathsf{In}).\ \mathsf{smash}(\mathsf{store}\ l\, v\, s, i', o)])\\ & (\mathcal{E}[\![E]\!] e\, s)(i)\end{aligned}$$

$$\begin{aligned}\mathcal{S}[\![\ \texttt{write}\ E\ ]\!] = {} & \lambda e \in \mathsf{Env}.\ \lambda(s \in \mathsf{S}, i \in \mathsf{In}, o \in \mathsf{Out}).\\ & (\lambda v \in \mathsf{SV}.\ \mathsf{smash}(s, i, \mathsf{extend}\ v\, o))\\ & (\mathcal{R}[\![E]\!] e\, s)\end{aligned}$$

$$\begin{aligned}\mathsf{extend} & = \lambda v \in \mathsf{SV}.\ [\lambda x \in \mathsf{O}.\ (v, \top),\\ & \qquad\qquad \lambda(v' \in \mathsf{SV}, o \in \mathsf{Out}).\ (v', \mathsf{extend}\ v\, o)]\\ & \in \mathsf{SV} \to \mathsf{Out} \to \mathsf{Out}\end{aligned}$$

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

$\mathsf{IO} = \mathsf{S} \oplus (\mathsf{SV} \multimap \mathsf{IO})_\perp \oplus (\mathsf{SV} \otimes \mathsf{IO}_\perp)$.

$\mathcal{P} : \textsc{Program} \to \mathsf{In} \to \mathsf{Out}$

$$\begin{aligned}
\mathcal{P}[\![\ \texttt{prog}\ S_1\ ]\!] = {} & \mathsf{fix}(\lambda h \in \mathsf{IO} \to \mathsf{In} \to \mathsf{Out}. \\
& \quad [\lambda s \in \mathsf{S}.\ \lambda i \in \mathsf{In}.\top, \\
& \quad \lambda f \in \mathsf{SV} \to \mathsf{IO}.\ \lambda(v \in \mathsf{SV}, i \in \mathsf{In}).\ h(f(v))(i), \\
& \quad \lambda(v \in \mathsf{SV}, io \in \mathsf{IO}).\ \lambda i \in \mathsf{In}.\ (v, \mathsf{up}(h(io)(i)))]) \\
& (\mathcal{S}[\![S_1]\!](\mathsf{void})(\mathsf{empty}))
\end{aligned}$$

$\mathcal{S} : \textsc{Statements} \to \mathsf{Env} \to \mathsf{S} \multimap \mathsf{IO}$

$$\begin{aligned}
\mathcal{S}[\![\ E_1\ \texttt{:=}\ E_2\ ]\!] = {} & \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}. \\
& (\lambda l \in \mathsf{Loc}.\ \lambda v \in \mathsf{SV}.\ \mathsf{in}_1(\mathsf{store}\ l\ v\ s)) \\
& (\mathcal{E}[\![E_1]\!]e\ s)(\mathcal{R}[\![E_2]\!]e\ s)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\![\ \texttt{read}\ E\ ]\!] = {} & \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}. \\
& (\lambda l \in \mathsf{Loc}.\ \mathsf{in}_2(\lambda v \in \mathsf{SV}.\ \mathsf{up}(\mathsf{in}_1(\mathsf{store}\ l\ v\ s)))) \\
& (\mathcal{E}[\![E_1]\!]e\ s)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\![\ \texttt{write}\ E\ ]\!] = {} & \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}. \\
& (\lambda v \in \mathsf{SV}.\ \mathsf{in}_3(v, \mathsf{up}(\mathsf{in}_1(s)))) \\
& (\mathcal{R}[\![E_2]\!]e\ s)
\end{aligned}$$

$\mathcal{S}[\![\ \texttt{skip}\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{id}_{\mathsf{S}}$

$$\begin{aligned}
\mathcal{S}[\![\ S_1\texttt{;}\ S_2\ ]\!] = {} & \lambda e \in \mathsf{Env}.\ \lambda s \in \mathsf{S}. \\
& \mathsf{fix}(\lambda g \in \mathsf{IO} \to \mathsf{IO}. \\
& \quad [\mathcal{S}[\![S_2]\!]e, \\
& \quad \lambda f \in \mathsf{SV} \to \mathsf{IO}.\ g \circ f, \\
& \quad \lambda(v \in \mathsf{SV}, io \in \mathsf{IO}).\ (v, \mathsf{up}(g(io)))]) \\
& (\mathcal{S}[\![S_1]\!]e\ s)
\end{aligned}$$

Table 2.33: Denotations for programs (interactive, direct)

$$\mathcal{G} : \text{GUARDED-STATEMENTS} \to \mathsf{Env} \to \mathsf{S} \circ\!\!\!\to (\mathsf{O} \oplus \mathsf{S}^{\natural})$$

$$\mathcal{G}[\![\ E \texttt{ -> } S_1\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$(\lambda t \in \mathsf{T}.\ \mathsf{if}\, t\, \mathsf{then\ in}_2(\mathcal{S}[\![S_1]\!]e\, s)\ \mathsf{else\ in}_1 \top)(\mathcal{R}[\![E]\!]e\, s)$$

$$\mathcal{G}[\![\ G_1\ \square\ G_2\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$[\lambda x \in \mathsf{O}.\ \mathsf{id}_{\mathsf{O}\oplus\mathsf{S}^{\natural}},$$
$$\lambda p_1 \in \mathsf{S}^{\natural}.\ [\lambda x \in \mathsf{O}.\ \mathsf{in}_2(p_1),$$
$$\lambda p_2 \in \mathsf{S}^{\natural}.\ \mathsf{in}_2(p_1 \uplus p_2)]](\mathcal{G}[\![G_1]\!]e\, s)(\mathcal{G}[\![G_2]\!]e\, s)$$

$$\mathcal{S} : \text{STATEMENTS} \to \mathsf{Env} \to \mathsf{S} \circ\!\!\!\to \mathsf{S}^{\natural}$$

$$\mathcal{S}[\![\ \texttt{if}\ G\ \texttt{fi}\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$[\lambda x \in \mathsf{O}.\ \{\!|s|\!\}, \mathsf{id}_{\mathsf{S}^{\natural}}](\mathcal{G}[\![G]\!]e\, s)$$

$$\mathcal{S}[\![\ \texttt{do}\ G\ \texttt{od}\ ]\!] = \lambda e \in \mathsf{Env}.\ \mathsf{fix}(\lambda c \in \mathsf{S} \circ\!\!\!\to \mathsf{S}^{\natural}.\ \mathsf{strict}\lambda s \in \mathsf{S}.$$
$$[\lambda x \in \mathsf{O}.\ \{\!|s|\!\}, \mathsf{ext}(c)](\mathcal{G}[\![G]\!]e\, s))$$

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;
y := 0;
do x=0  -> x := 1
[] x=0  -> y := y+1
od
```

```
x := 0;
y := 0;
do x=0  -> x := 1
[] x=0  -> y := y+1
[] 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

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.

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 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.

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

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.

[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.

[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.