3.4 Denotational Semantics

Denotational semantics, also known as *fixed-point semantics*, associates to each syntactically well-defined fragment of program a well-defined, rigorous *mathematical object*. This mathematical object denotes the complete behavior of the fragment of program, no matter in what context it will be used. In other words, the denotation of a fragment of program represents its contribution to the meaning of any program containing it. In particular, equivalence of programs or fragments is immediately translated into equivalence of mathematical objects. The later can be then shown using the entire arsenal of mathematics, which is supposedly better understood and more well-established than that of the relatively much newer field of programming languages. There are no theoretical requirements on the nature of the mathematical domains in which the fragments of program are interpreted, though a particular approach became quite well-established, to an extent that it is by many identified with denotational semantics itself: choose the domains to be appropriate bottomed complete partial orders (abbreviated BCPOs; see Section 2.9), and give the denotation of recursive language constructs (including loops, recursive functions, recursive data-structures, recursive types, etc.) as least fixed-points, which exist thanks to Theorem 1.

Each language requires customized denotational semantics, the same way each language required customized big-step or small-step structural operational semantics in the previous sections in this chapter. For the sake of concreteness, below we discuss general denotational semantics notions and notations by means of our running example language, IMP, without attempting to completely define it. Note that the complete denotational semantics of IMP is listed in Figure 3.20 in Section 3.4.1. Consider, for example, arithmetic expressions in IMP (which are side-effect free). Each arithmetic expression can be thought of as the mathematical object which is a partial function taking a state to an integer value, namely the value that the expression evaluates to in the given state. It is a partial function and not a total one because the evaluation of some arithmetic expressions may not be defined in some states, for example due to illegal operations such as division by zero. Thus, we can define the *denotation* of arithmetic expressions as a *total* function

$$[\![\_]\!] : \ AExp \to (State \rightharpoonup Int)$$

taking arithmetic expressions to *partial* functions from states to integer numbers. As in all the semantics discussed in this chapter, states are themselves partial maps from names to values. In what follows we will follow a common notational simplification and will write $[\![a]\!]\sigma$ instead of $[\![a]\!](\sigma)$ whenever $a \in AExp$ and $\sigma \in State$, and similarly for other syntactic or semantic categories. To avoid ambiguity in the presence of multiple denotation functions, many works on denotational semantics tag the denotation functions with their corresponding syntactic categories, e.g., $AExp[\![\_]\!]$ or $[\![\_]\!]_{AExp}$. Our IMP language is simple enough that we prefer not to add such tags.

Denotation functions are defined inductively, over the structure of the language constructs. For example, if $i \in Int$ then $[\![i]\!]$ is the constant function i, that is, $[\![i]\!]\sigma = i$ for any $\sigma \in State$. Similarly, if $x \in Id$ then $[\![x]\!]\sigma = \sigma(x)$. As it is the case in mathematics, if an undefined value is used to calculate another value, then the resulting value is also undefined. In particular, $[\![x]\!]\sigma$ is undefined when $x \notin Dom(\sigma)$. The denotation of compound constructs is defined in terms of the denotations of the parts. In other words, we say that denotational semantics is *compositional*. For example, $[\![a_1 + a_2]\!]\sigma = [\![a_1]\!]\sigma +_{Int} [\![a_2]\!]\sigma$ for any $a_1, a_2 \in AExp$ and any $\sigma \in State$. For the same reason as above, if any of $[\![a_1]\!]\sigma$ or $[\![a_2]\!]\sigma$ is undefined then $[\![a_1 + a_2]\!]\sigma$ is also implicitly undefined. One can also chose to explicitly keep certain functions undefined in certain states, such as the denotation of division in those states in which the denominator is zero:

$$[\![a_1 \,/\, a_2]\!]\sigma = \begin{cases} [\![a_1]\!]\sigma \;/_{\mathit{Int}}\; [\![a_2]\!]\sigma & \text{if } [\![a_2]\!]\sigma \neq 0 \\ \bot & \text{if } [\![a_2]\!]\sigma = 0 \end{cases}$$

Note that even though the case where $[\![a_2]\!]\sigma$ is undefined ($\bot$) was not needed to be explicitly listed above (because it falls under the first case, $[\![a_2]\!]\sigma \neq 0$), it is still the case that $[\![a_1 \,/\, a_2]\!]\sigma$ is undefined whenever any of $[\![a_1]\!]\sigma$ or $[\![a_2]\!]\sigma$ is undefined.

An immediate use of denotational semantics is to prove properties about programs. For example, we can show that the addition operation on *AExp* whose denotational semantics was given above is associative. Indeed, we can prove for any $a_1, a_2, a_3 \in \mathit{AExp}$ the following equality of partial functions

$$[\![(a_1 + a_2) + a_3]\!] = [\![a_1 + (a_2 + a_3)]\!]$$

using conventional mathematical reasoning and the fact that the sum $+_{\mathit{Int}}$ in the *Int* domain is associative (see Exercise 61). Note that denotational semantics allows us not only to prove properties about programs or fragments of programs relying on properties of their mathematical domains of interpretation, but also, perhaps even more importantly, it allows us to elegantly formulate such properties. Indeed, what does it mean for a language construct to be associative or, in general, for any desired property over programs or fragments of programs to hold? While one could use any of the operational semantics discussed in this chapter to answer this question, denotational semantics gives us one of the most direct means to state and prove program properties.

Each syntactic category is interpreted into its corresponding mathematical domain. For example, the denotations of Boolean expressions and of statements are total functions of the form:

$$\begin{aligned} [\![\_]\!] &: \mathit{BExp} \to (\mathit{State} \rightharpoonup \mathit{Bool}) \\ [\![\_]\!] &: \mathit{Stmt} \to (\mathit{State} \rightharpoonup \mathit{State}) \end{aligned}$$

The former is similar to the one for arithmetic expressions above, so we do not discuss it here. The latter is more interesting and deserves to be detailed. Statements can indeed be regarded as partial functions taking states into resulting states. In addition to partiality due to illegal operations in expressions that statements may involve, such as division by zero, partiality in the denotation of statements may also occur for another important reason: *loops may not terminate*. For example, the statement `while (`x`<=`y`) do skip` will not terminate in those states in which the value that x denotes is less than or equal to that of y. Mathematically, we say that the function from states to states that this loop statement denotes is undefined in those states in which the loop statement does not terminate. This will be elaborated shortly, after we discuss other statement constructs.

Since `skip` does not change the state, its denotation is the identity function, i.e., $[\![\texttt{skip}]\!] = 1_{\mathit{State}}$. The assignment statement updates the given state when defined in the assigned variable, that is, $[\![x := a]\!]\sigma = \sigma[[\![a]\!]\sigma/x]$ when $\sigma(x) \neq \bot$ and $[\![a]\!]\sigma \neq \bot$, and $[\![x := a]\!]\sigma = \bot$ otherwise. Sequential composition accumulates the state changes of the denotations of the composed statements, so it is precisely the mathematical composition of the corresponding partial functions: $[\![s_1 \,;\, s_2]\!] = [\![s_2]\!] \circ [\![s_1]\!]$.

As an example, let us calculate the denotation of the statement "`x := 1; y := 2; x := 3`" when `x` $\neq$ `y`, i.e., the function $[\![\texttt{x := 1; y := 2; x := 3}]\!]$. Applying the denotation of sequential composition twice, we obtain $[\![\texttt{x := 3}]\!] \circ [\![\texttt{y := 2}]\!] \circ [\![\texttt{x := 1}]\!]$. Applying this composed function on a state σ, one gets $([\![\texttt{x := 3}]\!] \circ [\![\texttt{y := 2}]\!] \circ [\![\texttt{x := 1}]\!])\sigma$ equals $\sigma[1/x][2/y][3/x]$ when $\sigma(x)$ and $\sigma(y)$ are both defined, and equals $\bot$ when any of $\sigma(x)$ or $\sigma(y)$ is undefined; let σ' denote

$\sigma[1/x][2/y][3/x]$. By the definition of function update, one can easily see that σ' can be defined as

$$\sigma'(z) = \begin{cases} 3 & \text{if } z = x \\ 2 & \text{if } z = y \\ \sigma(z) & \text{otherwise,} \end{cases}$$

which is nothing but $\sigma[2/y][3/x]$. We can therefore conclude that the statements "`x := 1; y := 2; x := 3`" and "`y := 2; x := 3`" are equivalent, because they have the same denotation.

The denotation of a conditional statement $\texttt{if}\, b\, \texttt{then}\, s_1\, \texttt{else}\, s_2$ in a state σ is either the denotation of s_1 in σ or that of s_2 in σ, depending upon the denotation of b in σ:

$$[\![\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \\ [\![s_2]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

The third case above was necessary, because the first two cases do not cover the entire space of possibilities and, in such situations, one may (wrongly in our context here) understand that the function is underspecified in the remaining cases rather than undefined. Using the denotation of the conditional statement above and conventional mathematical reasoning, we can show, for example, that $[\![$ `(if y <= z then x := 1 else x := 2); x := 3` $]\!]$ is the function taking states σ defined in x, y and z to $\sigma[3/x]$.

The language constructs which admit non-trivial and interesting denotational semantics tend to be those which have a recursive nature. One of the simplest such constructs, and the only one we discuss here (see Section 4.8 for other recursive constructs), is IMP's `while` looping construct. Thus, the question we address next is how to define the denotation functions of the form

$$[\![\texttt{while}\ b\ \texttt{do}\ s]\!] : \mathit{State} \rightharpoonup \mathit{State}$$

where $b \in \mathit{BExp}$ and $s \in \mathit{Stmt}$. What we want is $[\![\texttt{while}\ b\ \texttt{do}\ s]\!]\sigma = \sigma'$ iff the while loop correctly terminates in state σ' when executed in state σ. Such a σ' may not always exist for two reasons:

1. Because b or s is undefined (e.g., due to illegal operations) in σ or in other states encountered during the loop execution; or

2. Because s (which may contain nested loops) or the while loop itself does not terminate.

If w is the partial function $[\![\texttt{while}\ b\ \texttt{do}\ s]\!]$, then its most natural definition would appear to be:

$$w(\sigma) = \begin{cases} \sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ w([\![s]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{true} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

Mathematically speaking, this is a problematic definition for several reasons:

1. The partial function w is defined in terms of itself;

2. It is not clear that such a w exists; and

3. In case it exists, it is not clear that such a w is unique.

To see how easily one can yield inappropriate recursive definitions of functions, we refer the reader to the discussion immediately following Theorem 1, which shows examples of recursive definitions which admit no solutions or which are ambiguous.

We next develop the mathematical machinery needed to rigorously define and reason about partial functions like the w above. More precisely, we frame the mathematics needed here as an instance of the general setting and results discussed in Section 2.9. We strongly encourage the reader to familiarize herself with the definitions and results in Section 2.9 before continuing.

A convenient interpretation of partial functions that may ease the understanding of the subsequent material is as *information* or *knowledge bearers*. More precisely, a partial function $\alpha : State \rightharpoonup State$ can be thought of as carrying knowledge about some states in *State*, namely exactly those on which α is defined. For such a state $\sigma \in State$, the knowledge that α carries is $\alpha(\sigma)$. If α is not defined in a state $\sigma \in State$ then we can think of it as "α *does not have any information about* σ".

Recall from Section 2.9 that the set of partial functions between any two sets can be organized as a bottomed complete partial order (BCPO). In our case, if $\alpha, \beta : State \rightharpoonup State$ then we say that α *is less informative than or as informative as* β, written $\alpha \leq \beta$, if and only if for any $\sigma \in State$, it is either the case that $\alpha(\sigma)$ is not defined, or both $\alpha(\sigma)$ and $\beta(\sigma)$ are defined and $\alpha(\sigma) = \beta(\sigma)$. If $\alpha \leq \beta$ then we may also say that β *refines* α or that β *extends* α. Then $(State \rightharpoonup State, \leq \bot)$ is a BCPO, where $\bot : State \rightharpoonup State$ is the partial function which is undefined everywhere.

One can think of each possible iteration of a while loop as an opportunity to refine the knowledge about its denotation. Before the Boolean expression b of the loop $\texttt{while}\, b\, \texttt{do}\ s$ is evaluated the first time, the knowledge that one has about its denotation function w is the empty partial function $\bot :\ State \rightharpoonup State$, say w_0. Therefore, w_0 corresponds to no information.

Now suppose that we evaluate the Boolean expression b in some state σ and that it is false. Then the denotation of the while loop should return σ, which suggests that we can refine our knowledge about w from w_0 to the partial function $w_1 : State \rightharpoonup State$, which is an identity on all those states $\sigma \in State$ for which $[\![b]\!]\sigma = \texttt{false}$ and which remains undefined in any other state.

So far we have not considered any state in which the loop needs to evaluate its body. Suppose now that for some state σ, it is the case that $[\![b]\!]\sigma = \texttt{true}$, $[\![s]\!]\sigma = \sigma'$, and $[\![b]\!]\sigma' = \texttt{false}$, that is, that the while loop terminates in one iteration. Then we can extend w_1 to a partial function $w_2 : State \rightharpoonup State$, which, in addition to being an identity on those states on which w_1 is defined, that is $w_1 \leq w_2$, takes each σ as above to $w_2(\sigma) = \sigma'$.

By iterating this process, one can define a partial function $w_k : State \rightharpoonup State$ for any natural number k, which is defined on all those states on which the while loop terminates in *at most* k evaluations of its Boolean condition (i.e., $k - 1$ executions of its body). An immediate property of the partial functions w_0, w_1, w_2, ..., w_k is that they increasingly refine each other, that is, $w_0 \leq w_1 \leq w_2 \leq \cdots \leq w_k$. Informally, the partial functions w_k approximate w as k increases; more precisely, for any $\sigma \in State$, if $w(\sigma) = \sigma'$, that is, if the while loop terminates and σ' is the resulting state, then there is some k such that $w_k(\sigma) = \sigma'$. Moreover, $w_n(\sigma) = \sigma'$ for any $n \geq k$.

But the main question still remains unanswered: how to define the denotation $w : State \rightharpoonup State$ of the looping statement $\texttt{while}\, b\, \texttt{do}\ s$? According to the intuitions above, w should be some sort of *limit* of the (infinite) sequence of partial functions $w_0 \leq w_1 \leq w_2 \leq \cdots \leq w_k \leq \cdots$. We next formalize all the intuitions above. Let us define the total function

$$\mathcal{F} : (State \rightharpoonup State) \to (State \rightharpoonup State)$$

taking partial functions $\alpha : State \rightharpoonup State$ to partial functions $\mathcal{F}(\alpha) : State \rightharpoonup State$ as follows:

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \alpha([\![s]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{true} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

The partial functions w_k defined informally above can be now rigorously defined as $\mathcal{F}^k(\bot)$, where $\mathcal{F}^k$ stays for k compositions of $\mathcal{F}$, and $\mathcal{F}^0$ is by convention the identity function, i.e., $1_{(State \rightharpoonup State)}$ (which is total). Indeed, one can show by induction on k the following property, where $[\![s]\!]^i$ stays for i compositions of $[\![s]\!] : State \rightharpoonup State$ and $[\![s]\!]^0$ is by convention the identity (total) function on *State*:

$$\mathcal{F}^k(\bot)(\sigma) = \begin{cases} [\![s]\!]^i\sigma & \text{if there is } 0 \le i < k \text{ s.t. } [\![b]\!][\![s]\!]^i\sigma = \texttt{false} \\ & \text{and } [\![b]\!][\![s]\!]^j\sigma = \texttt{true} \text{ for all } 0 \le j < i \\ \bot & \text{otherwise} \end{cases}$$

We can also show that the following is a chain of partial functions (Exercise 65 requires the reader to prove, for the IMP language, all these facts mentioned above and below)

$$\bot \le \mathcal{F}(\bot) \le \mathcal{F}^2(\bot) \le \cdots \le \mathcal{F}^n(\bot) \le \cdots$$

in the BCPO $(State \rightharpoonup State, \le, \bot)$. As intuitively discussed above, this chain incrementally approximates the desired denotation of $\texttt{while}\, b \,\texttt{do}\, s$. The final step is to realize that $\mathcal{F}$ is a continuous function and thus satisfies the hypotheses of the fixed-point Theorem 1, so we can conclude that the least upper bound (lub) of the chain above, which by Theorem 1 is the least fixed-point $\mathit{fix}(\mathcal{F})$ of $\mathcal{F}$, is the desired denotation of the while loop, that is,

$$[\![\texttt{while}\ b\ \texttt{do}\ s]\!] = \mathit{fix}(\mathcal{F})$$

Remarks. First, note that we indeed want the *least* fixed-point of $\mathcal{F}$, and not some arbitrary fixed-point of $\mathcal{F}$, to be the denotation of the while statement. Indeed, any other fixed-points would define states in which the while loop is intended to be undefined. To be more concrete, consider the simple IMP while loop "`while not(k <= 10) do k := k + 1`" whose denotation is defined only on those states σ with $\sigma(k) \le 10$ and, on those states, it is the identity. That is,

$$[\![\texttt{while not(k <= 10) do k := k + 1}]\!](\sigma) = \begin{cases} \sigma & \text{if } \sigma(k) \le 10 \\ \bot & \text{otherwise} \end{cases}$$

Consider now another fixed-point $\gamma : State \rightharpoonup State$ of its corresponding $\mathcal{F}$. While γ must still be the identity on those states σ with $\sigma(k) \le 10$ (indeed, $\gamma(\sigma) = \mathcal{F}(\gamma)(\sigma) = \sigma$ for such $\sigma \in State$), it is not enforced to be undefined on any other states. In fact, it can be shown that the fixed-points of $\mathcal{F}$ are precisely those γ as above with the additional property that $\gamma(\sigma) = \gamma(\sigma')$ for any $\sigma, \sigma' \in State$ with $\sigma(k) > 10$, $\sigma'(k) > 10$, and $\sigma(x) = \sigma'(x)$ for any $x \ne k$. Such a γ can be, for example, the following:

$$\gamma(\sigma) = \begin{cases} \sigma & \text{if } \sigma(k) \le 10 \\ \iota & \text{otherwise} \end{cases}$$

where $\iota \in State$ is some arbitrary but fixed state. It is clear that such γ fixed-points are too informative for our purpose here, since we want the denotation of the while loop to be undefined in all states in which the loop does not terminate. Any other fixed-point of $\mathcal{F}$ which is strictly more informative that $\mathit{fix}(\mathcal{F})$ is simply too informative.

Second, note that the chain $\bot \leq \mathcal{F}(\bot) \leq \mathcal{F}^2(\bot) \leq \cdots \leq \mathcal{F}^n(\bot) \leq \cdots$ can be stationary in some cases, but in general it is not. For example, when the loop is well-defined and terminates in any state in some fixed maximum number of iterations which does not depend on the state, its denotation is the (total) function in which the chain stabilizes (which in that case is its lub and, by Theorem 1, the fixed-point of $\mathcal{F}$). For example, the chain corresponding to the loop "`while (1 <= k and k <= 10) do k := k + 1`" stabilizes in 12 steps, each step adding more states to the domain of the corresponding partial function until nothing can be added anymore: at step 1 all states σ with $\sigma(k) > 100$ or $\sigma(k) < 1$, at step 2 those with $\sigma(k) = 10$, at step 3 those with $\sigma(k) = 9$, ..., at step 11 those with $\sigma(k) = 1$; then no other state is added at step 12, that is, $\mathcal{F}^{12}(\bot) = \mathcal{F}^{11}(\bot)$. However, the chain associated to a loop is not stationary in general. For example, "`while (k <= 0) do k := k + 1`" terminates in any state, but there is no bound on the number of iterations. Consequently, there is no n such that $\mathcal{F}^n(\bot) = \mathcal{F}^{n+1}(\bot)$. Indeed, the later has strictly more information than the former: $\mathcal{F}^{n+1}$ is defined on all those states σ with $\sigma(k) = -n$, while $\mathcal{F}^n$ is not.

3.4.1 The Denotational Semantics of IMP

Figure 3.20 shows the complete denotational semantics of IMP. There is not much to comment on the denotational semantics of the various IMP language constructs, because they have already been discussed above. Note though that the denotation of conjunction captures the desired short-circuited semantics, in that the second conjunct is evaluated only when the first evaluates to `true`. Also, note that the denotation of programs is still a total function for uniformity (in spite of the fact that some programs may not be well-defined or may not terminate), but one into the BCPO $\mathit{State}_\bot$ (see Section 2.9); thus, the denotation of a program which is not well-defined is $\bot$. Finally, note that, like in the big-step SOS of IMP in Section 3.2.2, we ignore the non-deterministic evaluation strategies of the `+` and `/` arithmetic expression constructs. In fact, since the denotations of the various language constructs are *functions*, non-deterministic constructs cannot be handled in denotational semantics the same way they were handled in operational semantics. There are ways to deal with non-determinism and concurrency in denotational semantics as discussed at the end of this section, but those are more complex and lead to inefficient interpreters when executed, so we do not consider them in this book. We here limit ourselves to denotational semantics of deterministic languages.

3.4.2 Denotational Semantics in Equational/Rewriting Logic

In order to formalize and execute denotational semantics one needs to formalize and execute the fragment of mathematics that is used by the denotation functions. How much mathematics is used is open-ended and is typically driven by the particular programming language in question. Since a denotational semantics associates to each program or fragment of program a mathematical object expressed using the formalized language of the corresponding mathematical domain, the faithfulness of any representation/encoding/implementation of denotational semantics into any formalism directly depends upon the faithfulness of the formalizations of the mathematical domains.

The faithfulness of formalizations of mathematical domains is, however, quite hard to characterize in general. Each mathematical domain formalization may require its own proofs of correctness. Consider, for example, the basic domain of natural numbers. One may choose to formalize it using, e.g., Peano-style equational axioms or λ-calculus (see Section 4.5); nevertheless, none of

Arithmetic expression constructs

$[\![\_]\!] : AExp \to (State \rightharpoonup Int)$

$[\![i]\!]\sigma = i$

$[\![x]\!]\sigma = \sigma(x)$

$[\![a_1 + a_2]\!]\sigma = [\![a_1]\!]\sigma \ +_{Int} \ [\![a_2]\!]\sigma$

$$[\![a_1 \,/\, a_2]\!]\sigma = \begin{cases} [\![a_1]\!]\sigma \ /_{Int} \ [\![a_2]\!]\sigma & \text{if } [\![a_2]\!]\sigma \neq 0 \\ \bot & \text{if } [\![a_2]\!]\sigma = 0 \end{cases}$$

Boolean expression constructs

$[\![\_]\!] : BExp \to (State \rightharpoonup Bool)$

$[\![t]\!]\sigma = t$

$[\![a_1 \texttt{<=} a_2]\!]\sigma = [\![a_1]\!]\sigma \ \leq_{Int} \ [\![a_2]\!]\sigma$

$[\![\texttt{not}\, b]\!]\sigma = \neg_{Bool}([\![b]\!]\sigma)$

$$[\![b_1 \,\texttt{and}\, b_2]\!]\sigma = \begin{cases} [\![b_2]\!]\sigma & \text{if } [\![b_1]\!]\sigma = \texttt{true} \\ \texttt{false} & \text{if } [\![b_1]\!]\sigma = \texttt{false} \\ \bot & \text{if } [\![b_1]\!]\sigma = \bot \end{cases}$$

Statement constructs

$[\![\_]\!] : Stmt \to (State \rightharpoonup State)$

$[\![\texttt{skip}]\!]\sigma = \sigma$

$$[\![x \,\texttt{:=}\, a]\!]\sigma = \begin{cases} \sigma[[\![a]\!]\sigma \ / \ x] & \text{if } \sigma(x) \neq \bot \\ \bot & \text{if } \sigma(x) = \bot \end{cases}$$

$[\![s_1 \,\texttt{;}\, s_2]\!]\sigma = [\![s_2]\!][\![s_1]\!]\sigma$

$$[\![\,\texttt{if}\, b \,\texttt{then}\, s_1 \,\texttt{else}\, s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \\ [\![s_2]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

$[\![\,\texttt{while}\, b \,\texttt{do}\, s]\!] = \mathit{fix}(\mathcal{F})$, where $\mathcal{F} : (State \rightharpoonup State) \to (State \rightharpoonup State)$ defined as

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \alpha([\![s]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{true} \\ \sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

Programs

$[\![\_]\!] : Pgm \to State_\bot$

$[\![\texttt{var}\, xl \,\texttt{;}\, s]\!] = [\![s]\!](xl \mapsto 0)$

Figure 3.20: Denot(IMP): The denotational semantics of IMP.

these formalizations is powerful enough to mechanically derive any property over natural numbers. We therefore cannot prove faithfulness theorems for our representation of denotational semantics in equational/rewriting logic as we did for the other semantic approaches. Instead, we here limit ourselves to demonstrating our approach using the concrete IMP language and the basic domains of integer numbers and Booleans, together with the mathematical domain already formalized in Section 2.9.5, which allows us to define and execute higher-order functions and fixed-points for them.

Denotational Semantics of IMP in Equational/Rewrite Logic

Figure 3.21 shows a direct representation of the denotational semantics of IMP in Figure 3.20 using the mathematical domain of higher-order functions and fixed-points for them that we formalized in rewriting logic (actually in its membership equational fragment) in Section 2.9.5.

To reduce the number of denotation functions defined, or in other words to simplify our denotational semantics, we chose to collapse all the syntactic categories under only one sort, *Syntax*. Similarly, to reuse existing operations of the CPO domain in Section 2.9.5 (e.g., the substitution) on the new mathematical domains without any additional definitional effort, we collapse all the mathematical domains under CPO; in other words, we now have only one large mathematical domain, CPO, which includes the domains of integer numbers, Booleans and states as subdomains.

There is not much to say about the equations in Figure 3.21; they restate the mathematical definitions in Figure 3.20 using our particular CPO language. The equational formalization of the CPO domain in Section 2.9.5 propagates undefinedness through the CPO operations according to their evaluation strategies. This allows us to ignore some cases, such as the last case in the denotation of the conditional in Figure 3.20; indeed, if $\texttt{app}_{CPO}([\![A]\!], \sigma)$ is undefined in some state σ then $[\![X := A]\!]\sigma$ will also be undefined, because $\texttt{app}_{CPO}(..., \bot)$ equals $\bot$ according to our CPO formalization in Section 2.9.5.

☆ Denotational Semantics of IMP in Maude

Figure 3.22 shows the Maude module corresponding to the rewrite theory in Figure 3.21.

3.4.3 Notes

Denotational semantics is the oldest semantic approach to programming languages. The classic paper that introduced denotational semantics as we know it today, then called "mathematical semantics", was published in 1971 by Scott and Strachey [80]. However, Strachey's interest in the subject started much earlier; he published two papers in 1966 and 1967, [88] and [89], respectively, which mark the beginning of denotational semantics. Before Strachey and Scott published their seminal paper [80], Scott published in 1970 a paper which founded what we call today *domain theory* [81]. Initially, Scott formalized domains as complete latices (which also admit a fixed-point theorem); in time, bottomed complete partial orders (BCPOs, see Section 2.9) turned out to have better properties and they eventually replaced the complete latices.

We have only used very simple domains in our semantics of IMP in this section, such as domains of integers, Booleans, and partial functions. Moreover, for simplicity, we are not going to use complex domains for the IMP++ extension in Section 3.5 either. However, complex languages or better semantics may require more complex domains. In fact, choosing the right domains is one of the most important aspects of a denotational semantics. Poor domains may lead to behavioral

sort:

$Syntax$ // generic sort for syntax

subsorts:

$AExp,\ BExp,\ Stmt,\ Pgm\ <\ Syntax$ // syntactic categories fall under $Syntax$

$Int,\ Bool,\ State\ <\ CPO$ // basic domains are regarded as CPOs

operation:

$[\![\_]\!] : Syntax \to [\,CPO\,]$ // denotation of syntax

equations:

// Arithmetic expression constructs:

$[\![I]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ I$

$[\![X]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \sigma(X)$

$[\![A_1 + A_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ (\mathtt{app}_{CPO}([\![A_1]\!], \sigma)\ +_{Int}\ \mathtt{app}_{CPO}([\![A_2]\!], \sigma))$

$$[\![A_1\ /\ A_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{if}_{CPO}(\ \mathtt{app}_{CPO}([\![A_2]\!], \sigma)\ \mathtt{=/=}_{Bool}\ 0, \\ \mathtt{app}_{CPO}([\![A_1]\!], \sigma)\ /_{Int}\ \mathtt{app}_{CPO}([\![A_2]\!], \sigma),\ \ \bot)$$

// Boolean expression constructs:

$[\![T]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ T$

$[\![A_1\ \mathtt{<=}\ A_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ (\mathtt{app}_{CPO}([\![A_1]\!], \sigma)\ \leq_{Int}\ \mathtt{app}_{CPO}([\![A_2]\!], \sigma))$

$[\![\,\mathtt{not}\ B]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ (\mathtt{not}_{Bool}\ \mathtt{app}_{CPO}([\![B]\!], \sigma))$

$[\![B_1\ \mathtt{and}\ B_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{if}_{CPO}(\mathtt{app}_{CPO}([\![B_1]\!], \sigma),\ \mathtt{app}_{CPO}([\![B_2]\!], \sigma),\ \mathtt{false}\,)$

// Statement constructs:

$[\![\,\mathtt{skip}\,]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \sigma$

$[\![X\ \mathtt{:=}\ A]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{app}_{CPO}(\ \mathtt{fun}_{CPO}\ arg\ \mathtt{->}\ \mathtt{if}_{CPO}(\sigma(X) \neq \bot,\ \sigma[arg/X],\ \bot), \mathtt{app}_{CPO}([\![A]\!], \sigma))$

$[\![S_1\ ;\ S_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{app}_{CPO}([\![S_2]\!], \mathtt{app}_{CPO}([\![S_1]\!], \sigma))$

$[\![\,\mathtt{if}\ B\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{if}_{CPO}(\mathtt{app}_{CPO}([\![B]\!], \sigma), \mathtt{app}_{CPO}([\![S_1]\!], \sigma), \mathtt{app}_{CPO}([\![S_2]\!], \sigma))$

$$[\![\,\mathtt{while}\ B\ \mathtt{do}\ S]\!] = \mathtt{fix}_{CPO}(\ \mathtt{fun}_{CPO}\ \alpha\ \mathtt{->}\ \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->} \\ \mathtt{if}_{CPO}(\mathtt{app}_{CPO}([\![B]\!], \sigma), \mathtt{app}_{CPO}(\alpha, \mathtt{app}_{CPO}([\![S]\!], \sigma)), \sigma))$$

// Programs:

$[\![\,\mathtt{var}\ Xl\ ;\ S]\!] = \mathtt{app}_{CPO}([\![S]\!],\ (Xl \mapsto 0))$

Figure 3.21: $\mathcal{R}_{\text{DENOT(IMP)}}$: Denotational semantics of IMP in equational/rewriting logic.

```
mod IMP-SEMANTICS-DENOTATIONAL is including IMP-SYNTAX + STATE + CPO .
  sort Syntax .
  subsorts AExp BExp Stmt Pgm < Syntax .
  subsorts Int Bool State < CPO .

  op [[_]] : Syntax  -> CPO .    --- Syntax interpreted in CPOs

  var X : Id . var Xl : List{Id} .  var I : Int .  var A1 A2 A : AExp .
  var T : Bool .  var B1 B2 B : BExp .  var S1 S2 S : Stmt .
  ops sigma alpha arg : -> CPOVar .

  eq [[I]] = funCPO sigma -> I .

  eq [[X]] = funCPO sigma -> sigma(X) .

  eq [[A1 + A2]] = funCPO sigma -> (appCPO([[A1]],sigma) +Int appCPO([[A2]],sigma)) .

  eq [[A1 / A2]] = funCPO sigma -> ifCPO(appCPO([[A2]],sigma) =/=Bool 0,
                                        appCPO([[A1]],sigma) /Int appCPO([[A2]],sigma),
                                        undefined) .

  eq [[T]] = funCPO sigma -> T .

  eq [[A1 <= A2]] = funCPO sigma -> (appCPO([[A1]],sigma) <=Int appCPO([[A2]],sigma)) .

  eq [[not B]] = funCPO sigma -> (notBool appCPO([[B]],sigma)) .

  eq [[B1 and B2]] = funCPO sigma -> ifCPO(appCPO([[B1]],sigma),appCPO([[B2]],sigma),false) .

  eq [[skip]] = funCPO sigma -> sigma .

  eq [[X := A]]
   = funCPO sigma
     -> appCPO(funCPO arg
               -> ifCPO(sigma(X) =/=Bool undefined, sigma[arg / X], undefined),
               appCPO([[A]],sigma)) .

  eq [[S1 ; S2]] = funCPO sigma -> appCPO([[S2]],appCPO([[S1]],sigma)) .

  eq [[if B then S1 else S2]]
   = funCPO sigma -> ifCPO(appCPO([[B]],sigma),appCPO([[S1]],sigma),appCPO([[S2]],sigma)) .

  eq [[while B do S]]
   = fixCPO(funCPO alpha
            -> funCPO sigma
               -> ifCPO(appCPO([[B]],sigma),appCPO(alpha,appCPO([[S]],sigma)),sigma)) .

  eq [[(var Xl ; S)]] = appCPO([[S]],(Xl |-> 0)) .
endm
```

Figure 3.22: The denotational semantics of IMP in Maude

limitations or to non-modular denotational semantic definitions. There are two additional important contributions to domain theory that are instrumental in making denotational semantics more usable:

- *Continuation domains.* The use of *continuations* in denotational semantics was proposed in 1974, in a paper by Strachey and Wadsworth [90]. Wadsworth was the one who coined the term "continuation", as representing "the meaning of the rest of the program". Continuations allow to have direct access to the execution flow, in particular to modify it, as needed for the semantics of abrupt termination, exceptions, or call/cc (Scheme was the first language to support call/cc). This way, continuations bring modularity and elegance to denotational definitions. However, they come at a price: using continuations affects the entire language definition (so one needs to change almost everything) and the resulting semantics are harder to read and reason about. There are countless uses of continuations in the literature, not only in denotational semantics; we refer the interested reader to a survey paper by Reynolds, which details continuations and their discovery from various perspectives [73].

- *Powerdomains.* The usual domains of partial functions that we used in our denotational semantics of IMP are not sufficient to define non-deterministic and/or concurrent languages. Consider, for example, the denotation of statements, which are partial functions from states to states. If the language is non-deterministic or concurrent, then a statement may take a state into any of many possible different states, or, said differently, it may take a state into a *set* of states. To give denotational semantics to such languages, Plotkin proposed and formalized the notion of *powerdomain* [69]; the elements of a powerdomain are sets of elements of an underlying domain. Powerdomains make it thus possible to give denotational semantics to non-deterministic languages; used in combination with a technique called "resumptions", powerdomains can also be used to give interleaving semantics to concurrent languages.

As seen above, most of the denotational semantics ideas and principles have been proposed and developed in the 1970s. While it is always recommended to read the original papers for historical reasons, some of them may actually use notational conventions and constructions which are not in current use today, making them less accessible. The reader interested in a more modern presentation of denotational semantics and domain theory is referred to Schmidt's denotational semantics book [79], and to Mosses' denotational semantics chapter [59] and Gunter and Scott's domain theory chapter [34] in the Handbook of Theoretical Computer Science (1990).

Denotational semantics are commonly defined and executed using functional languages. A particularly appealing aspect of functional languages is that the domains of partial functions, which are crucial for almost any denotational semantics, and fixed-point operators for them can be very easily defined using the already existing functional infrastructure of these languages; in particular, one needs to define no λ-like calculus as we did in Section 2.9.5. There is a plethora of works on implementing and executing denotational semantics on functional languages. We here only mention Papaspyrou's denotational semantics of C [67] which is implemented in Haskell; it uses about 40 domains in total and spreads over about 5000 lines of Haskell code. An additional advantage of defining denotational semantics in functional languages is that one can relatively easily port them into theorem provers and then prove properties or meta-properties about them. We refer the interested reader to Nipkow [64] for a simple example of how this is done in the context of the Isabelle/HOL prover.

A very insightful exercise is to regard domain theory and denotational semantics through the lenses of *initial algebra semantics* [30], which was proposed by Goguen *et al.* in 1977. The initial algebra semantics approach is simple and faithful to equational logic (Section 2.3); it can be summarized with the following steps:

1. Define a language syntax as an algebraic signature, say Σ, which admits an initial (term) algebra, say T_Σ;

2. Define any semantic domain of interest as the carrier of corresponding sort in some (domain) algebra, say D;

3. Give D a Σ-algebra structure, by defining operations corresponding to all symbols in Σ;

4. Conclude that the meaning of the language in D is the unique morphism $D_{\_} : T_\Sigma \to D$, which gives meaning D_t in D to any fragment of program t.

Let us apply the initial algebra semantics steps above to IMP:

1. Σ is the signature in Figure 3.2;

2. D_{AExp} is the BCPO $(\mathit{State} \rightharpoonup \mathit{Int}, \preceq, \bot)$ and similarly for the other sorts;

3. $D_{\_+\_} : D_{AExp} \times D_{AExp} \to D_{AExp}$ is the (total) function defined as $D_{\_+\_}(f_1, f_2)(\sigma) = f_1(\sigma) +_{\mathit{Int}} f_2(\sigma)$ for all $f_1, f_2 \in D_{AExp}$ and $\sigma \in \mathit{State}$, and similarly for the other syntactic constructs.

4. The meaning of IMP is given by the unique morphism $D_{\_} : T_\Sigma \to D_{AExp}$.

Therefore, one can regard a denotational semantics of a language as an initial algebra semantics applied to a *particular* algebra D; in our IMP case, for example, what we defined as $[\![a]\!]$ for $a \in AExp$ in denotational semantics is nothing but D_a. Moreover, we can now *prove* our previous denotational semantics definitions; for example, we can prove $D_{a_1 + a_2}(\sigma) = D_{a_1}(\sigma) +_{\mathit{Int}} D_{a_2}(\sigma)$. The above was a very brief account of initial algebra semantics, but sufficient to appreciate the foundational merits of initial algebra semantics in the context of denotational semantics (initial algebra semantics has many other applications). From an initial algebra semantics perspective, a denotational semantics is all about defining an algebra D in which the syntax is interpreted. How each fragment gets a meaning follows automatically, from more basic principles. In that regard, initial algebra semantics achieves a cleaner separation of concerns: syntax is defined as a signature, and semantics is defined as an algebra. There are no equations mixing syntax and semantics, like $[\![a_1 + a_2]\!] = [\![a_1]\!] +_{\mathit{Int}} [\![a_2]\!]$.

We are not aware of any other approaches to define denotational semantics using rewriting and then executing it on rewrite engines as we did in Section 3.4.2. While this is not difficult in principle, as seen in this section, it requires one to give executable rewriting definitions of semantic domains and of fixed points. This may be a tedious and repetitive process on simplistic rewrite engines; for example, the use of membership equational logic, which allows computations to take place also on terms whose intermediate sorts cannot be determined, was crucial for our formalization in Section 2.9.5. Perhaps the closest approach to ours is the one by Goguen and Malcolm in [33]; they define a simple imperative language using the OBJ system (a precursor of Maude) and a style which is a mixture of initial algebra semantics and operational semantics. For example, no fixed-points are used in [33], the loops being simply unrolled like in small-step SOS (see Section 3.3).

3.4.4 Exercises

Prove the following exercises, all referring to the IMP denotational semantics in Figure 3.20.

Exercise 61. *Show the associativity of the addition expression construct, that is, that*

$$[\![(a_1 \texttt{+} a_2) \texttt{+} a_3]\!] = [\![a_1 \texttt{+} (a_2 \texttt{+} a_3)]\!]$$

for any $a_1, a_2, a_3 \in AExp$.

Exercise 62. *Show the associativity of the statement sequential composition, that is, that*

$$[\![s_1 \texttt{;} (s_2 \texttt{;} s3)]\!] = [\![(s_1 \texttt{;} s_2) \texttt{;} s_3]\!]$$

for any $s_1, s_2, s_3 \in Stmt$. *Compare the elegance of formulating and proving this result using denotational semantics with the similar task using small-step SOS (see Exercise 55).*

Exercise 63. *State and prove the (correct) distributivity property of division over addition.*

Exercise 64. *Prove the equivalence of statements of the form "*$(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ \texttt{;}\ s$*" and "*$\texttt{if}\ b\ \texttt{then}\ (s_1\ \texttt{;}\ s)\ \texttt{else}\ (s_2\ \texttt{;}\ s)$*".*

Exercise* 65. *Prove that the functions* $\mathcal{F} : (State \rightharpoonup State) \to (State \rightharpoonup State)$ *associated to* IMP *while loops satisfy the hypotheses of the fixed-point Theorem 1, so that the denotation of* IMP *loops is indeed well-defined. Also, prove that the partial functions* $w_k : State \rightharpoonup State$ *defined as*

$$w_k(\sigma) = \begin{cases} [\![s]\!]^i\sigma & \text{if there is } 0 \le i < k \text{ s.t. } [\![b]\!][\![s]\!]^i\sigma = \texttt{false} \text{ and } [\![b]\!][\![s]\!]^j\sigma = \texttt{true} \text{ for all } 0 \le j < i \\ \bot & \text{otherwise} \end{cases}$$

are well-defined, that is, that if an i *as above exists then it is unique. Then prove that* $w_k = \mathcal{F}^k(\bot)$.