Chapter 4

Denotational Semantics

4.1 The Denotational Semantics Game

We have seen how an operational semantics is a natural tool for evaluating programs and proving properties like termination. However, it is less than ideal for many purposes. A framework based on transitions between configurations of an abstract machine is usually better suited for reasoning about complete programs than program fragments. In PostFix, for instance, it is necessary to extend the operational semantics with an elaborate notion of observational equivalence and transform equivalence in order to effectively demonstrate the interchangeability of command sequences. Additionally, the emphasis on syntactic entities in an operational semantics can complicate reasoning. For example, syntactically distinct executable sequence answers in PostFix must be treated as the same observable value in order to support a non-trivial notion of observational equivalence for command sequences. Finally, the step-by-step nature of an operational semantics can suggest notions of time and dependency that are not essential to the language being defined. For example, an operational semantics for the expression language EL might specify that the left operand of a binary operator is evaluated before the right even though this difference may be impossible to detect in practice.

An alternate framework for reasoning about programs is suggested by the notion of transform equivalence developed for PostFix. According to this notion, each PostFix command sequence is associated with a stack transform that describes how the sequence maps an input stack to an output stack. It is natural to view these stack transforms as functions. For example, the stack transform associated with the command sequence [`3`, `add`] would be an *add3* function with the following graph:[1]

$$\{\langle \textit{error-stack}, \textit{error-stack}\rangle,\ \langle [\,], \textit{error-stack}\rangle, \ldots,\\ \langle [-1], [2]\rangle,\ \langle [0], [3]\rangle,\ \langle [1], [4]\rangle, \ldots,\\ \langle [\textit{add3}], \textit{error-stack}\rangle,\ \langle [\textit{mul2}], \textit{error-stack}\rangle, \ldots,\\ \langle [5,\ 23], [8,\ 23]\rangle,\ \langle [5,\ \textit{mul2},\ 17,\ \textit{add3}], [8,\ \textit{mul2},\ 17,\ \textit{add3}]\rangle, \ldots\}$$

Here, *error-stack* stands for a distinguished error stack analogous to S_{error} in the extended PostFix SOS. Stack elements that are executable sequences are represented by their stack transforms (e.g., *add3* and *mul2*) rather than some syntactic phrase.

Associating stack transform functions with command sequences has several benefits. First, this perspective directly supports a notion of equivalence for program phrases. For example, the *add3* function is the stack transform associated with the sequence $[\texttt{1}, \texttt{add}, \texttt{2}, \texttt{add}]$ as well as the sequence $[\texttt{3}, \texttt{add}]$. This implies that the two sequences are behaviorally indistinguishable and can be safely interchanged in any PostFix context. The fact that stack elements that are executable sequences are represented by functions rather than syntactic entities greatly simplifies this kind of reasoning.

The other major benefit of this approach is that the stack transform associated with the concatenation of two sequences is easily composed from the stack transforms of the component sequences. For example, suppose that the sequence $[\texttt{2}, \texttt{mul}]$ is modelled by the *mul2* function, whose graph is sketched below:

$$\{\langle \textit{error-stack}, \textit{error-stack}\rangle,\ \langle [\,], \textit{error-stack}\rangle, \ldots,\\ \langle [-1], [-2]\rangle,\ \langle [0], [0]\rangle,\ \langle [1], [2]\rangle, \ldots,\\ \langle [\textit{add3}], \textit{error-stack}\rangle,\ \langle [\textit{mul2}], \textit{error-stack}\rangle, \ldots,\\ \langle [5,\ 23], [10,\ 23]\rangle,\ \langle [5,\ \textit{mul2},\ 17,\ \textit{add3}], [10,\ \textit{mul2},\ 17,\ \textit{add3}]\rangle, \ldots\}$$

Then the stack transform of $[\texttt{3}, \texttt{add}, \texttt{2}, \texttt{mul}] \equiv [\texttt{3}, \texttt{add}] \mathbin{@} [\texttt{2}, \texttt{mul}]$ is simply the function $\textit{mul2} \circ \textit{add3}$, whose graph is:

$$\{\langle \textit{error-stack}, \textit{error-stack}\rangle,\ \langle [\,], \textit{error-stack}\rangle, \ldots,\\ \langle [-1], [4]\rangle,\ \langle [0], [6]\rangle,\ \langle [1], [8]\rangle, \ldots,\\ \langle [\textit{add3}], \textit{error-stack}\rangle,\ \langle [\textit{mul2}], \textit{error-stack}\rangle, \ldots,\\ \langle [5,\ 23], [16,\ 23]\rangle,\ \langle [5,\ \textit{mul2},\ 17,\ \textit{add3}], [16,\ \textit{mul2},\ 17,\ \textit{add3}]\rangle, \ldots\}$$

Similarly the stack transform of $[\texttt{2}, \texttt{mul}, \texttt{3}, \texttt{add}] \equiv [\texttt{2}, \texttt{mul}] \mathbin{@} [\texttt{3}, \texttt{add}]$ is the function $\textit{add3} \circ \textit{mul2}$, whose graph is:

[1]Here, and for the rest of this chapter, we rely heavily on the metalanguage concepts and notations described in Appendix A. Consult this appendix as necessary to unravel the formalism.

$$\begin{aligned}&\{\langle \textit{error-stack}, \textit{error-stack}\rangle,\ \langle [\,], \textit{error-stack}\rangle, \ldots,\\ &\ \langle[-1], [1]\rangle,\ \langle[0], [3]\rangle,\ \langle[1], [5]\rangle, \ldots,\\ &\ \langle[\textit{add3}], \textit{error-stack}\rangle,\ \langle[\textit{mul2}], \textit{error-stack}\rangle, \ldots,\\ &\ \langle[5,\ 23], [13,\ 23]\rangle,\ \langle[5,\ \textit{mul2},\ 17,\ \textit{add3}], [13,\ \textit{mul2},\ 17,\ \textit{add3}]\rangle, \ldots\}\end{aligned}$$

The notion that the meaning of a program phrase can be determined from the meaning of its parts is the essence of a framework called **denotational semantics**. A denotational semantics determines the meaning of a phrase in a compositional way based on its static structure rather than on some sort of dynamically changing configuration. Unlike an operational semantics, a denotational semantics emphasizes *what* the meaning of a phrase is, not *how* the phrase is evaluated.

The basic structure of the denotational framework is illustrated in Figure 4.1. A denotational semantics consists of three parts:

1. A **syntactic algebra** that describes the abstract syntax of the language under study. This can be specified by the s-expression grammar approach introduced in Chapter 2.

2. A **semantic algebra** that models the meaning of program phrases. A semantic algebra consists of a collection of semantic domains along with functions that manipulate these domains. The meaning of a program may be something as simple as an element of a primitive semantic domain like *Int*, the domain of integers. More typically, the meaning of a program is an element of a function domain that maps **context domains** to an **answer domain**, where

    - Context domains are the denotational analog of state components in an SOS configuration. They model such entities as name/value associations, the current contents of memory, and control information.
    - An answer domain represents the possible meanings of programs. In addition to a component that models what we would normally think of as being the result of a program phrase, the answer domain may also include components that model context information that was transformed by the program.

3. A **meaning function** that maps elements of the syntactic algebra (i.e., nodes in the abstract syntax trees) to their meanings in the semantic algebra. Each phrase is said to **denote** its image under the meaning function. Not any function can serve as a meaning function; the function must be a homomorphism between the syntactic algebra and the semantic algebra.

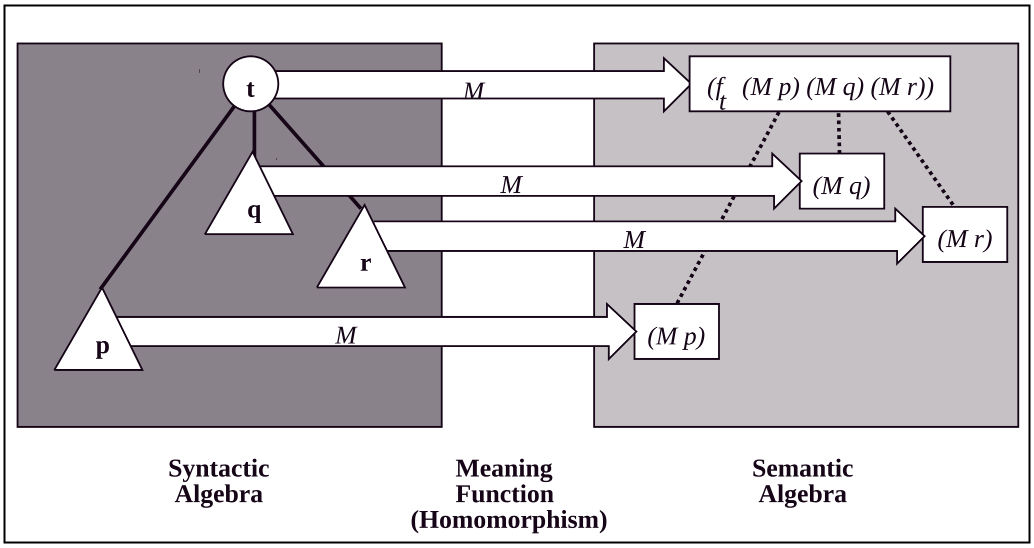

Figure 4.1: The denotational semantics "game board".

> This is just the technical condition that constrains the meaning of an abstract syntax tree node to be determined from the meaning of its subnodes. It can be stated more formally as follows:
>
> > Suppose M is a meaning function and t is a node in an abstract syntax tree, with children $t_1, \ldots, t_k$. Then
> >
> > $$(M\ \ t) \text{ must equal } (f_t\ \ (M\ \ t_1)\ \ldots\ (M\ \ t_k)\ \ )$$
> >
> > where f_t is a function that is determined by the syntactic class of t.

The advantage of restricting meaning functions to homomorphisms is that their structure-preserving behavior greatly simplifies reasoning. This design choice accounts for the compositional nature of denotational semantics (i.e., the meaning of the whole is determined by the meaning of the parts).

4.2 A Denotational Semantics for PostFix

As an example of the denotational framework, we will flesh out the details of the denotational description of PostFix that were sketched above. The abstract syntax for PostFix was already provided in Figure 2.7, so the syntactic algebra is already taken care of. We therefore need to construct the semantic algebra and the meaning function.

$$
\begin{array}{lll}
t & \in & \mathit{Stack\text{-}Transform} \;=\; \mathit{Stack} \to \mathit{Stack} \\
s & \in & \mathit{Stack} \;=\; \mathit{Value}^* + \mathit{Error} \\
v & \in & \mathit{Value} \;=\; \mathit{Int} + \mathit{Stack\text{-}Transform} \\
a & \in & \mathit{Answer} \;=\; \mathit{Value} + \mathit{Error} \\
 & & \mathit{Error} \;=\; \{\mathrm{error}\} \\
i & \in & \mathit{Int} \;=\; \{\ldots,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots\} \\
b & \in & \mathit{Bool} \;=\; \{\mathit{true}, \mathit{false}\}
\end{array}
$$

Figure 4.2: Semantic domains for the PostFix denotational semantics

4.2.1 A Semantic Algebra for PostFix

What kind of mathematical entities should we use to model PostFix programs? Suppose that we have some sort of entity representing stacks. Then it's natural to model both PostFix commands and command sequences as functions that transform one stack entity into another. For example, the `swap` command could be modelled by a function that takes a stack as an argument, and returns a stack in which the top two elements have been swapped.

We need to make some provision for the case where the stack contains an insufficient number of elements or the wrong type of elements. For this purpose we will assume that there is a distinguished stack, *error-stack*, that indicates that an error has occurred. For example, calling the transform associated with the `swap` command on a stack with less than two elements should return *error-stack*. All transforms should return *error-stack* when given *error-stack* as an argument.

Figure 4.2 presents domain equations that describe one implementation of this approach. The *Stack-Transform* domain consists of functions from stacks to stacks, where an element of the domain *Stack* is either a sequence of values or the distinguished error stack (here modelled by the single element of the unit domain *Error*). The domain *Value* of stackable values includes not only integers but also stack transforms, which model executable sequences that have been pushed on the stack. Since PostFix programs can signal errors, the domain *Answer* of answers includes an error value in addition to the stackable values.

A somewhat unsettling property of the domain equations in the figure is that they are defined recursively — transforms operate on stacks, which themselves may contain transforms. In Chapter 7 we will discuss how to understand a set of recursively defined equations. For now, we'll just assume that these equations have a sensible interpretation.

We extend the semantic domains into a semantic algebra by defining a collection of functions that manipulate the domains. Right now we'll just specify the interfaces to these functions. We'll defer the details of their definitions until after we've studied the meaning function. This will allow us to move more quickly to the core of the denotational semantics — the meaning function — without getting sidetracked by various issues concerning the definition of the semantic functions.

Figure 4.3 gives informal specifications for the functions that we will use to manipulate the semantic domains. We will defer studying the implementation of these functions until later. *empty-stack*, *error-stack*, and *error-answer* are just names for useful constants. *push*, *pop*, and *top* are the usual stack operations. Their specifications are complicated somewhat by the details of error handling. For example, *top* returns an element of *Answer* rather than just *Value* because it must return *error-answer* in the case where the given stack is empty. *push* takes its argument from *Answer* rather than *Value* so that it can be composed with *top*. *arithop* is an auxiliary function that will simplify the meaning function for arithmetic and relational commands. It serves to abstract over common behavior (replacing the top two integers on the stack by some value that depends on them) while suppressing error detail (return an error stack if any error is encountered along the way). *transform* facilitates error handling when an answer that's expected to be a transform turns out to be an integer or an error answer instead.

The signatures of the functions, especially the stack functions, may seem strange at first glance, because none of them explicitly refer to the *Stack* domain. But recall that *Stack-Transform* is defined to be $Stack \rightarrow Stack$, so that the signature of *push*, for instance, is really

$$Answer \rightarrow (Stack \rightarrow Stack)$$

From this perspective, *push* probably seems more familiar: it is a function that takes an answer and stack (in curried form) and returns a stack. However, since stack transforms are the key abstraction of this semantics, we have written the signatures to emphasize this fact. Under this view, *push* is a function that takes an answer and returns a stack transform. Of course, in either case *push* is exactly the same mathematical structure; the only difference is in how we think about it!

4.2.2 A Meaning Function for PostFix

Now we're ready to study the meaning function for PostFix. What we call the meaning function is actually a collection of so-called **valuation functions**, one for each syntactic domain defined by the abstract syntax for the language. For each syntactic domain, the name of the associated valuation function is usually a script version of the metavariable that ranges over that domain. For example,

- *empty-stack* : *Stack*
  The empty stack of values.
- *error-stack* : *Stack*
  The distinguished error stack.
- *error-transform* : *Stack-Transform*
  A transform that maps all stacks to *error-stack*.
- *error-answer* : *Answer*
  An error in the domain *Answer*.
- *push* : *Answer* $\rightarrow$ *Stack-Transform*
  Given the answer value v, return a transform that pushes v on a stack; otherwise return *error-transform*.
- *pop* : *Stack-Transform*
  For a nonempty stack s, return the stack resulting from popping the top value; otherwise return *error-stack*.
- *top* : *Stack* $\rightarrow$ *Answer*
  Given a nonempty stack s, return an answer of the top element of s; otherwise return an *error-answer*.
- *arithop* : (*Int* $\rightarrow$ *Int* $\rightarrow$ *Answer*) $\rightarrow$ *Stack-Transform*
  Let $f : \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Answer}$ be the functional argument to *arithop*. Return a transform with the following behavior: if the given stack has two integers i_1 and i_2 followed by s_{rest}, then return a stack whose top value v_{result} is followed by s_{rest}, where $(\mathit{Value} \mapsto \mathit{Answer}\ v_{\mathrm{result}})$ is the answer resulting from the application $(f\ \ i_2\ \ i_1)$. If the given stack is not of this form or if the result of applying f is *error-answer*, then return *error-stack*.
- *transform* : *Answer* $\rightarrow$ *Stack-Transform*
  Given an answer that is a stack transform, return it; otherwise return the error transform.

Figure 4.3: Specifications for functions manipulating the semantic domains for PostFix

$\mathcal{P}$: Program $\rightarrow$ *Answer*
$\mathcal{Q}$: Commands $\rightarrow$ *Stack-Transform*
$\mathcal{C}$: Command $\rightarrow$ *Stack-Transform*
$\mathcal{A}$: Arithmetic-operator $\rightarrow$ (*Int* $\rightarrow$ *Int* $\rightarrow$ *Answer*)
$\mathcal{R}$: Relational-operator $\rightarrow$ (*Int* $\rightarrow$ *Int* $\rightarrow$ *Bool*)
$\mathcal{N}$: Intlit $\rightarrow$ *Int*

Figure 4.4: Signatures of the PostFix valuation functions.

$\mathcal{P}$ is the valuation function for $P \in$ Program, $\mathcal{C}$ is the valuation function for $C \in$ Command, and so on.

The signatures of valuation functions contain valuable information about the meaning of the language. It is always prudent to study the signatures before delving into the details of the definitions for the valuation functions.

The signatures for the PostFix valuation functions appear in Figure 4.4. In the case of PostFix, one of the things the signatures say is that programs map to answers,

$$\mathcal{P} : \text{Program} \rightarrow \textit{Answer},$$

where an answer (as defined above) is either an error or a stackable value. Since the *Value* domain contains both integers and stack transforms, either of these is a legal return value; PostFix is not limited to returning only integers. Also, the fact that *Answer* contains an error value indicates that it's possible for the program to signal an error, a rather important fact. If the signature of $\mathcal{P}$ were instead

$$\mathcal{P} : \text{Program} \rightarrow \textit{Value},$$

then this signature would tell us that errors could not be signalled, since the *Value* domain does not contain any element modelling an error.

The signatures also tell us that both commands and command sequences map to stack transforms. Since stack transforms are easily composable, this suggests that the meaning of a command sequence will be some sort of composition of the meanings of its component commands. This turns out to be the case. Finally, the return types of $\mathcal{A}$ and $\mathcal{R}$ remind us, respectively, of the arguments types to the *arithop* and *relop*, two of the auxiliary functions specified in Figure 4.3. This is more than coincidence; in fact, the valuation functions and auxiliary functions were designed to dovetail in a nice way.

$$\mathcal{P}[\![\texttt{(}Q\texttt{)}]\!] = (top\ (\mathcal{Q}[\![Q]\!]\ empty\text{-}stack))$$

$$\mathcal{Q}[\![C\ .\ Q]\!] = \mathcal{Q}[\![Q]\!] \circ \mathcal{C}[\![C]\!]$$

$$\mathcal{Q}[\![]\!] = \lambda s\ .\ s$$

$$\mathcal{C}[\![N]\!] = (push\ (Value \mapsto Answer\ (Int \mapsto Value\ \mathcal{N}[\![N]\!])))$$

$$\mathcal{C}[\![\texttt{(}Q\texttt{)}]\!] = (push\ (Value \mapsto Answer\ (Stack\text{-}Transform \mapsto Value\ \mathcal{Q}[\![Q]\!])))$$

$$\mathcal{C}[\![\texttt{pop}]\!] = pop$$

$$\mathcal{C}[\![\texttt{swap}]\!] = \lambda s\ .\ (push\ (top\ (pop\ s))\ (push\ (top\ s)\ (pop\ (pop\ s))))$$

$$\begin{aligned}\mathcal{C}[\![\texttt{sel}]\!] = \lambda s\ .\ & \textbf{matching}\ (top\ (pop\ (pop\ s)))\\ & \triangleright\ (Value \mapsto Answer\ (Int \mapsto Value\ i))\ [\!]\\ & \quad (push\ (\textbf{if}\ (i =_{\text{Int}} 0)\ \textbf{then}\ (top\ s)\ \textbf{else}\ (top\ (pop\ s))\ \textbf{fi})\\ & \qquad\quad (pop\ (pop\ (pop\ s)))\ )\\ & \triangleright\ \textbf{else}\ error\text{-}stack\quad \textbf{endmatching}\end{aligned}$$

$$\mathcal{C}[\![\texttt{exec}]\!] = \lambda s\ .\ (transform\ (top\ s)\ (pop\ s))$$

$$\mathcal{C}[\![A]\!] = (arithop\ \mathcal{A}[\![A]\!])$$

$$\begin{aligned}\mathcal{C}[\![R]\!] = (arithop\ (\lambda i_1 i_2\ .\ & (Value \mapsto Answer\\ & (Int \mapsto Value\ (\textbf{if}\ (\mathcal{R}[\![R]\!]\ i_1\ i_2)\ \textbf{then}\ 1\ \textbf{else}\ 0\ \textbf{fi})))))\end{aligned}$$

$$\mathcal{A}[\![\texttt{sub}]\!] = \lambda i_1 i_2\ .\ (Value \mapsto Answer\ (Int \mapsto Value\ (i_1 -_{\text{Int}} i_2)))$$

Similarly for `add`, `mul`

$$\begin{aligned}\mathcal{A}[\![\texttt{div}]\!] = \lambda i_1 i_2\ .\ & \textbf{if}\ (i_2 =_{\text{Int}} 0)\ \textbf{then}\ error\text{-}answer\\ & \textbf{else}\ (Value \mapsto Answer\ (Int \mapsto Value\ (i_1\ /_{\text{Int}}\ i_2)))\ \textbf{fi}\end{aligned}$$

$$\mathcal{R}[\![\texttt{lt}]\!] = <_{\text{Int}}$$

Similarly for `eq` and `gt`

$\mathcal{N}$ maps integer numerals to the integer numbers they denote.

Figure 4.5: The valuation functions for PostFix.

Now we're ready to study the PostFix valuation functions themselves, which appear in Figure 4.5. The meaning of a program consisting of command sequence Q is the top of the stack that results from applying the transform associated with Q to the empty stack. Error cases are handled correctly because *top* yields an error answer when given an empty stack or error stack. The meaning of a command sequence is the composition of the transforms of its component commands. The order of the composition

$$\mathcal{Q}[\![Q]\!] \circ \mathcal{C}[\![C]\!] = \lambda s \,.\, (\mathcal{Q}[\![Q]\!] \ (\mathcal{C}[\![C]\!] \ s))$$

is crucial, because it guarantees that the stack manipulations of the first command can be observed by the subsequent commands. Reversing the order of the composition would have the effect of executing commands in a right-to-left order instead. The stack transform associated with the empty command sequence is the identity function on stacks.

Most of the clauses for the command valuation clause $\mathcal{C}$ are straightforward. The integers and transforms corresponding to numerals and executable sequences are simply pushed onto the stack after an appropriate injection into the *Value* domain.[2] The transforms for `pop` and `swap` are simple compositions of auxiliary stack functions. The `sel` transform selects one of the top two stack elements based on the numeric value of the third stack element; an error is signalled if the third element is not an integer. In the `exec` transform, the top stack element is expected to be a stack transform t representing an executable sequence. Applying t to the rest of the stack yields the stack resulting from executing the executable sequence. If the top stack element is not a stack transform, an error is signalled. The meaning of arithmetic and relational commands is determined by *arithop* in conjunction with $\mathcal{A}$ and $\mathcal{R}$, valuation functions that map operator symbols like `add` and `lt` to the expected functions and predicates. $\mathcal{A}$ treats `div` specially so that division by 0 signals an error.

Before we move on, a few notes about reading denotational definitions are in order. Valuation functions tend to be remarkably elegant and concise. But this does not mean that they are always easy to read! To the contrary, the density of information in a denotational definition often demands meticulous attention from the reader. The ability to read semantic functions and valuation functions is a skill that requires patient practice to acquire. At first, unravelling such a

[2]Whereas the operational semantics used a stack with syntactic values — integer numerals and command sequences — the denotational semantics uses a stack of semantic values — integers and stack transforms. This is because the valuation functions $\mathcal{N}$ and $\mathcal{Q}$ are readily available for translating the syntactic elements to the semantic ones. Here and elsewhere we will follow the convention of using explicit injections in denotational descriptions.

definition may seem like solving a puzzle or doing detective work. However, the time invested in reading definitions of this sort pays off handsomely in terms of deep insights into the meanings of programming languages.

The conciseness of a denotational definition is due in large part to the liberal use of higher-order functions, i.e., functions that take other functions as arguments or return them as results. *arithop* is an excellent example of such a function: it takes an argument in the function domain $Int \rightarrow Int \rightarrow Value$, and returns a stack transform, which itself is an element of the function domain $Stack \rightarrow Stack$.

Definitions involving higher-order functions can be rather daunting to read until you acquire a knack for them. A typical problem is to think that pieces are missing. For example, a common reaction to the valuation clause for numerals,

$$\mathcal{C}[\![N]\!] = (push\ \ (Value \mapsto Answer\ \ (Int \mapsto Value\ \ \mathcal{N}[\![N]\!])))$$

is that a stack is somehow missing. After all, the value has to be pushed onto *something* — where is it? Carefully considering types, however, will show that nothing is missing. Recall that the signature of *push* is $Answer \rightarrow Stack\text{-}Transform$. Since

$$(Value \mapsto Answer\ \ (Int \mapsto Value\ \ \mathcal{N}[\![N]\!]))$$

is clearly an element of *Answer*, the result of the *push* application is a stack transform. Since $\mathcal{C}$ is supposed to map commands to stack transforms, the definition is well-typed. It's possible to introduce an explicit stack in this valuation clause by wrapping the right hand side in a λ of a stack argument:

$$\mathcal{C}[\![N]\!] = \lambda s\ .\ (push\ \ (Value \mapsto Answer\ \ (Int \mapsto Value\ \ \mathcal{N}[\![N]\!]))\ \ s)$$

This form of the definition probably seems much more familiar, because it's more apparent that the meaning of the command is a function that takes a stack and returns a stack, and *push* is actually given a stack on which to push its value. But the two definitions are equivalent. In order to stress the power of higher-order functions, we will continue to use the more concise versions. We encourage you to type check the definitions and expand them with extra λs as ways of improving your skill at reading them.

4.2.3 Semantic Functions for PostFix: the Details

Now that we've studied the core of the PostFix semantics, we'll flesh out the details of the functions specified in Figure 4.3. Figure 4.6 presents an implementation of the specifications. As an exercise, you should make sure that these definitions type check, and that they satisfy the specifications in Figure 4.3.

$$
\begin{array}{l}
\mathit{empty\text{-}stack} : \mathit{Stack} = (\mathit{Value}^* \mapsto \mathit{Stack}\ [\,]_{\mathit{Value}}) \\
\text{error-stack} : \mathit{Stack} = (\mathit{Error} \mapsto \mathit{Stack}\ \text{error}) \\
\text{error-transform} : \mathit{Stack\text{-}Transform} = \lambda s\ .\ \text{error-stack} \\
\text{error-answer} : \mathit{Answer} = (\mathit{Error} \mapsto \mathit{Answer}\ \text{error}) \\
\\
\mathit{push} : \mathit{Answer} \to \mathit{Stack\text{-}Transform} \\
= \lambda a\, s\ .\ \textbf{matching}\ a \\
\qquad \triangleright (\mathit{Value} \mapsto \mathit{Answer}\ v)\ \|\ \textbf{matching}\ s \\
\qquad\qquad\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ v^*)\ \|\ (\mathit{Value}^* \mapsto \mathit{Stack}\ (v\,.\,v^*)) \\
\qquad\qquad\qquad \triangleright\, \textbf{else}\ \mathit{error\text{-}stack}\ \ \textbf{endmatching} \\
\qquad \triangleright (\mathit{Error} \mapsto \mathit{Answer}\ \text{error})\ \|\ \mathit{error\text{-}stack}\ \ \textbf{endmatching} \\
\\
\mathit{pop} : \mathit{Stack\text{-}Transform} \\
= \lambda s\ .\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ v^*)\ \| \\
\qquad\ \ \textbf{if}\ (\text{empty?}\ v^*)\ \ \textbf{then}\ \text{error-stack}\ \textbf{else}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ (\mathit{tail}\ \ v^*))\ \ \textbf{fi} \\
\qquad \triangleright\, \textbf{else}\ \text{error-stack}\ \ \textbf{endmatching} \\
\\
\mathit{top} : \mathit{Stack} \to \mathit{Answer} \\
= \lambda s\ .\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ v^*)\ \| \\
\qquad\ \ \textbf{if}\ (\text{empty?}\ v^*)\ \ \textbf{then}\ \text{error-answer}\ \textbf{else}\ (\mathit{Value} \mapsto \mathit{Answer}\ (\text{head}\ \ v^*))\ \ \textbf{fi} \\
\qquad \triangleright\, \textbf{else}\ \text{error-answer}\ \ \textbf{endmatching} \\
\\
\mathit{arithop} : (\mathit{Int} \to \mathit{Int} \to \mathit{Answer}) \to \mathit{Stack\text{-}Transform} \\
= \lambda f\, s\ .\ \textbf{matching}\ (\mathit{top}\ \ s) \\
\qquad \triangleright (\mathit{Int} \mapsto \mathit{Value}\ \ i_1)\ \|\ \textbf{matching}\ (\mathit{top}\ \ (\mathit{pop}\ \ s)) \\
\qquad\qquad\qquad \triangleright (\mathit{Int} \mapsto \mathit{Value}\ \ i_2)\ \|\ (\mathit{push}\ \ (f\ \ i_2\ \ i_1)\ \ (\mathit{pop}\ \ (\mathit{pop}\ \ s))) \\
\qquad\qquad\qquad \triangleright\, \textbf{else}\ \mathit{error\text{-}stack}\ \ \textbf{endmatching} \\
\qquad \triangleright\, \textbf{else}\ \mathit{error\text{-}stack}\ \ \textbf{endmatching} \\
\\
\text{transform} : \mathit{Answer} \to \mathit{Stack\text{-}Transform} \\
= \lambda a\ .\ \textbf{matching}\ a \\
\qquad \triangleright (\mathit{Value} \mapsto \mathit{Answer}\ \ (\mathit{Stack\text{-}Transform} \mapsto \mathit{Value}\ \ t))\ \|\ \ t \\
\qquad \triangleright\, \textbf{else}\ \text{error-transform}\ \ \textbf{endmatching}
\end{array}
$$

Figure 4.6: Functions manipulating the semantic domains for PostFix.

Notice that several functions in Figure 4.6 describe similar manipulations. *push*, *pop*, and *arithop* all check to see if their input stack is an error stack. If so, they return *error-stack*; if not, they perform some manipulation on the sequence of values in the stack.

As in programming, in denotational semantics it is good practice to create abstractions that capture common patterns of behavior. Whenever the same pattern appears in multiple places within a denotational definition, it's a good idea to extract that pattern and box it up into a new entity. This can improve the clarity of the definitions while at the same time making them more compact.

We illustrate this kind of abstraction by introducing two higher-order functions that simplify the semantic functions for PostFix (see Figure 4.7.):

- *with-stack-values* takes a function f from value sequences to stacks and returns a stack transform that
  - maps a non-error stack to the result of applying f to the value sequence in the stack, and
  - maps an error stack to an error stack.
- *with-integer&stack* takes a function f from an integer to a stack transform and returns a stack transform that
  - maps any stack whose value sequence consists of an integer i followed by s_{rest} to the result of applying f to i and s_{rest}, and
  - maps any stack not of this form to the error stack.

The purpose of these new functions is to hide the details of error handling in order to highlight more important manipulations. As shown in Figure 4.7, rewriting *push* and *pop* in terms of *with-stack-values* removes an error check from each definition. Using *with-integer&stack* greatly simplifies *arithop*; the updated version concisely captures its essence without the distraction of case analyses and error checks.

As with the valuation functions, these highly condensed semantic functions can be challenging for the uninitiated to read. The fact that *push*, *pop*, and *arithop* are ultimately manipulating a stack is even harder to see in the new versions than it was in the original ones. As suggested before, reasoning about types and inserting extra λs can help. For example, since the result of a call to *with-integer&stack* is a stack transform t, and t is equivalent to $\lambda s \,.\, (t\ s)$, the new version of *arithop* can be rewritten as:

$$
\begin{array}{l}
\mathit{with\text{-}stack\text{-}values} : (\mathit{Value}^* \to \mathit{Stack}) \to \mathit{Stack\text{-}Transform} \\
= \lambda f\, s\ .\ \ \textbf{matching}\ s \\
\qquad\qquad \triangleright\, (\mathit{Value}^* \mapsto \mathit{Stack}\ \ v^*)\ [\!]\ \ (f\ \ v^*) \\
\qquad\qquad \triangleright\, \textbf{else}\ \mathit{error\text{-}stack}\ \ \textbf{endmatching} \\
\\
\mathit{with\text{-}integer\&stack} : (\mathit{Int} \to \mathit{Stack\text{-}Transform}) \to \mathit{Stack\text{-}Transform} \\
= \lambda f\ .\ \ (\mathit{with\text{-}stack\text{-}values} \\
\qquad\qquad (\lambda v^*\ .\ \ \textbf{matching}\ v^* \\
\qquad\qquad\qquad\qquad \triangleright\, (\mathit{Int} \mapsto \mathit{Value}\ \ i)\ .\ s_{\mathrm{rest}}\ [\!]\ \ (f\ \ i\ \ s_{\mathrm{rest}}) \\
\qquad\qquad\qquad\qquad \triangleright\, \textbf{else}\ \mathit{error\text{-}stack}\ \ \textbf{endmatching}\ )) \\
\\
\mathit{push} : \mathit{Answer} \to \mathit{Stack\text{-}Transform} \\
= \lambda a\ .\ \ \textbf{matching}\ a \\
\qquad\qquad \triangleright\, (\mathit{Value} \mapsto \mathit{Answer}\ \ v)\ [\!]\ \ (\mathit{with\text{-}stack\text{-}values}\ \ (\lambda v^*\ .\ (\mathit{Value}^* \mapsto \mathit{Stack}\ \ (v\ .\ v^*)))) \\
\qquad\qquad \triangleright\, (\mathit{Error} \mapsto \mathit{Answer}\ \ \mathrm{error})\ [\!]\ \ \mathit{error\text{-}transform} \\
\qquad\qquad \textbf{endmatching} \\
\\
\mathit{pop} : \mathit{Stack\text{-}Transform} \\
= (\mathit{with\text{-}stack\text{-}values} \\
\qquad (\lambda v^*\ .\ \ \textbf{if}\ (\mathrm{empty?}\ \ v^*)\ \ \textbf{then}\ \mathit{error\text{-}stack}\ \textbf{else}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ \ (\mathit{tail}\ \ v^*))\ \ \textbf{fi})) \\
\\
\mathit{arithop} : (\mathit{Int} \to \mathit{Int} \to \mathit{Answer}) \to \mathit{Stack\text{-}Transform} \\
= \lambda f\ .\ \ (\mathit{with\text{-}integer\&stack} \\
\qquad\qquad (\lambda i_1\ .\ \ (\mathit{with\text{-}integer\&stack} \\
\qquad\qquad\qquad\qquad (\lambda i_2\ .\ \ (\mathit{push}\ \ (f\ \ i_2\ \ i_1)))))) \\
\end{array}
$$

Figure 4.7: The auxiliary functions *with-stack-values* and *with-integer&stack* simplify some of the semantic functions for PostFix. (Only the modified functions are shown.)

$$
\begin{array}{l}
\lambda f\, s_0\, .\ ((\mathit{with\text{-}integer\&stack} \\
\qquad\quad (\lambda i_1 s_1\, .\ ((\mathit{with\text{-}integer\&stack} \\
\qquad\qquad\qquad\qquad (\lambda i_2 s_2\, .\ (\mathit{push}\ (f\ \ i_2\ \ i_1)\ \ s_2))) \\
\qquad\qquad\qquad\quad s_1))) \\
\qquad s_0)
\end{array}
$$

At least in this form it's easier to see that there are stacks from which each occurrence of *with-integer&stack* can extract an integer and substack.

Even more important is recognizing the pattern

$$((\mathit{with\text{-}integer\&stack}\ \ (\lambda i\, s_{\mathrm{rest}}\, .\, E))\ \ s)$$

as a construct that binds names to values. One way to pronounce this pattern is:

> "Let i be the top value of s and s_{rest} be all but the top value of s in the expression E. Return the value of E, except when s is empty or its top value isn't an integer, in which cases the error stack should be returned instead."

One of the powers of lambda notation is that it supports the invention of new binding constructs like *with-integer&stack* via higher-order functions without requiring any new syntactic extensions to the metalanguage. We will make extensive use of this power to simplify our future denotational definitions. Later we will see how this idea appears in programming under the name **continuation passing style** (Chapter 11).

4.3 Using the PostFix Denotational Semantics

The denotational definition of PostFix presented in the previous section may be mathematically elegant, but how do we use such a semantics to tell us anything useful about the language? In this section, we demonstrate the utility of the denotational definition in two areas:

1. Finding the meanings of PostFix programs.
2. Proving the observational equivalence of PostFix command sequences.

We will see that reasoning about individual programs can be more cumbersome in the denotational framework than in the operational framework. Yet, the compositional structure of the denotational semantics makes it more amenable to proving certain properties than the operational semantics.

4.3.1 The Meaning of PostFix Programs

Under the denotational approach, the meaning of a program is determined by the valuation function corresponding to a program node in the abstract syntax tree. Consider the sample PostFix program

```
((5 sub) 3 swap exec)
```

Because $\mathcal{P}$ maps programs to their meanings, $\mathcal{P}[\![$`((5 sub) 3 swap exec)`$]\!]$ *is* the meaning of this program. However, this fact is not very useful as stated because the element of *Answer* denoted by the program is not immediately apparent from the form of the metalanguage expression $\mathcal{P}[\![$`((5 sub) 3 swap exec)`$]\!]$.

We would like to massage the metalanguage expression for the meaning of a program into another metalanguage expression more recognizable as an element of the answer domain. We do this by using **equational reasoning** to simplify the metalanguage expression. That is, we are allowed to make any simplifications that are allowed by usual mathematical reasoning about the entities denoted by the metalanguage expressions. Equational reasoning allows us to perform such manipulations as:

- substitute equals for equals,
- apply functions to arguments,
- equate two function-denoting expressions when, for each argument, they map that argument to the same result (this is called **extensionality**).

Instances of equational reasoning are organized into **equational proofs** that contain a series of equalities. Figure 4.8 presents an equational proof that $\mathcal{P}[\![$`((5 sub) 3 swap exec)`$]\!]$ is equal to $(\mathit{Value} \mapsto \mathit{Answer}\ (\mathit{Int} \mapsto \mathit{Value}\ \text{-}2))$, where the latter metalanguage expression is immediately recognizable as an element of *Answer*. Each equality in the proof is justified by familiar mathematical rules. For example, the equality

$$\begin{aligned}&(\mathit{top}\ (((\mathcal{Q}\ [\![\texttt{3 swap exec}]\!]) \circ (\mathcal{C}\ [\![\texttt{(5 sub)}]\!]))\ \mathit{empty\text{-}stack}))\\ &= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{3 swap exec}]\!]\ (\mathcal{C}\ [\![\texttt{(5 sub)}]\!]\ \mathit{empty\text{-}stack})))\end{aligned}$$

is justified by the definition of function composition, while the equality

$$\begin{aligned}&(\mathit{top}\ (\mathcal{Q}\ [\![\texttt{swap exec}]\!]\ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Answer}\ v_2)\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1]))))\\ &= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{swap exec}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2, v_1])))\end{aligned}$$

is justified by the specification for the *push* function. The final expression in Figure 4.8 clearly denotes the integer -2, appropriately injected into the *Answer* domain. Since all the intermediate reasoning steps are equalities, we can conclude that

$$\mathcal{P}[\![\texttt{((5 sub) 3 swap exec)}]\!] = (\mathit{Value} \mapsto \mathit{Answer}\ (\mathit{Int} \mapsto \mathit{Value}\ \text{-}2))$$

We reiterate that $\mathcal{P}[\![\texttt{((5 sub) 3 swap exec)}]\!]$, as well as every intermediate expression, stands for the same mathematical entity as $(\mathit{Value} \mapsto \mathit{Answer}\ (\mathit{Int} \mapsto \mathit{Value}\ \text{-}2))$. The whole purpose of an equational proof is to simplify the original expression into a metalanguage expression whose form more directly expresses the structure of that entity.

Just as programs can be simplified by introducing procedural abstractions, equational proofs can often be simplified by structuring them more hierarchically. In the case of proofs, the analog to a programming language procedure is a theorem. For example, it's not difficult to prove a theorem (see the next section) that states that for any numeral N, any command sequence Q, and any stack s, that the following equality is valid:

$$(\mathcal{Q}\ [\![N\ .\ Q]\!]\ s) = (\mathcal{Q}\ [\![Q]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [(\mathit{Int} \mapsto \mathit{Value}\ \mathcal{N}[\![N]\!])\ .\ s]))$$

This theorem is analogous to the operational rewrite rule for handling integer numeral commands. It can be used to justify equalities like

$$\begin{aligned}&(\mathit{top}\ (\mathcal{Q}\ [\![\texttt{3 swap exec}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1])))\\ &= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{swap exec}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [(\mathit{Int} \mapsto \mathit{Value}\ 3), v_1])))\end{aligned}$$

A few such theorems can greatly reduce the length of the sample evaluation sequence. In fact, if we prove other theorems analogous to the operational rules, we can obtain an evaluation sequence that closely corresponds to the configuration sequence for an operational execution of the program (see Figure 4.9).

An advantage of the denotational framework is that equational proofs need not exhibit the step-by-step execution flavor of an operational execution. Since the steps of the proof are equalities, *any* equational reasoning techniques can be applied in *any* order. For example, Figure 4.10 shows a rather different approach to reasoning about the sample program. Here, the emphasis is on composing the meaning of the command sequence from its component commands. The sequence makes use of the following straightforward theorem

$$(\mathcal{Q}\ [\![C_1\ \ C_2\ \ \ldots C_n]\!]) = (\mathcal{C}\ [\![C_n]\!]) \circ \ldots \circ (\mathcal{C}\ [\![C_2]\!]) \circ (\mathcal{C}\ [\![C_1]\!])$$

as well as the following equality (not proved here):

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{((5 sub) 3 swap exec)}]\!] \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{(5 sub) 3 swap exec}]\!]\ \mathit{empty\text{-}stack})) \\
= (\mathit{top}\ (((\mathcal{Q}\ [\![\texttt{3 swap exec}]\!]) \circ (\mathcal{C}\ [\![\texttt{(5 sub)}]\!]))\ \mathit{empty\text{-}stack})) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{3 swap exec}]\!]\ (\mathcal{C}\ [\![\texttt{(5 sub)}]\!]\ \mathit{empty\text{-}stack}))) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{3 swap exec}]\!]\ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Answer}\ v_1)\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\,])))) \\
\quad \text{where } v_1\ = (\mathit{Stack\text{-}Transform} \mapsto \mathit{Value}\ (\mathcal{Q}\ [\![\texttt{(5 sub)}]\!])) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{3 swap exec}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1]))) \\
= (\mathit{top}\ (((\mathcal{Q}\ [\![\texttt{swap exec}]\!]) \circ (\mathcal{C}\ [\![\texttt{3}]\!]))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1]))) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{swap exec}]\!]\ (\mathcal{C}\ [\![\texttt{3}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1])))) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{swap exec}]\!]\ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Answer}\ v_2)\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1])))) \\
\quad \text{where } v_2\ = (\mathit{Int} \mapsto \mathit{Value}\ (\mathcal{N}\ [\![\texttt{3}]\!]))\ = (\mathit{Int} \mapsto \mathit{Value}\ 3) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{swap exec}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2, v_1]))) \\
= (\mathit{top}\ (((\mathcal{Q}\ [\![\texttt{exec}]\!]) \circ (\mathcal{C}\ [\![\texttt{swap}]\!]))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2, v_1]))) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{exec}]\!]\ (\mathcal{C}\ [\![\texttt{swap}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2, v_1])))) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{exec}]\!]\ (\mathit{push}\ (\mathit{top}\ (\mathit{pop}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2, v_1]))) \\
\qquad\qquad\qquad\qquad (\mathit{push}\ (\mathit{top}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2, v_1])) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{pop}\ (\mathit{pop}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2, v_1]))))))) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{exec}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1, v_2]))) \\
= (\mathit{top}\ (((\mathcal{Q}\ [\![\,]\!]) \circ (\mathcal{C}\ [\![\texttt{exec}]\!]))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1, v_2]))) \\
= (\mathit{top}\ (((\lambda s\ .\ s) \circ (\mathcal{C}\ [\![\texttt{exec}]\!]))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1, v_2]))) \\
= (\mathit{top}\ (\mathcal{C}\ [\![\texttt{exec}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1, v_2]))) \\
= (\mathit{top}\ (\mathit{transform}\ (\mathit{top}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1, v_2]))\ (\mathit{pop}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_1, v_2])))) \\
= (\mathit{top}\ (\mathit{transform}\ (\mathit{Value} \mapsto \mathit{Answer}\ v_1)\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2]))) \\
= (\mathit{top}\ (\mathit{transform}\ (\mathit{Value} \mapsto \mathit{Answer}\ (\mathit{Stack\text{-}Transform} \mapsto \mathit{Value}\ (\mathcal{Q}\ [\![\texttt{(5 sub)}]\!]))) \\
\qquad\qquad\qquad (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2]))) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{5 sub}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2]))) \\
= (\mathit{top}\ (((\mathcal{Q}\ [\![\texttt{sub}]\!]) \circ (\mathcal{C}\ [\![\texttt{5}]\!]))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2]))) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{sub}]\!]\ (\mathcal{C}\ [\![\texttt{5}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2])))) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{sub}]\!]\ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Answer}\ v_3)\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_2])))) \\
\quad \text{where } v_3\ = (\mathit{Int} \mapsto \mathit{Value}\ (\mathcal{N}\ [\![\texttt{5}]\!]))\ = (\mathit{Int} \mapsto \mathit{Value}\ 5) \\
= (\mathit{top}\ (\mathcal{Q}\ [\![\texttt{sub}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_3, v_2]))) \\
= (\mathit{top}\ (((\mathcal{Q}\ [\![\,]\!]) \circ (\mathcal{C}\ [\![\texttt{sub}]\!]))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_3, v_2]))) \\
= (\mathit{top}\ (((\lambda s\ .\ s) \circ (\mathcal{C}\ [\![\texttt{sub}]\!]))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_3, v_2]))) \\
= (\mathit{top}\ (\mathcal{C}\ [\![\texttt{sub}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_3, v_2]))) \\
= (\mathit{top}\ (\mathit{arithop}\ (\mathcal{A}\ [\![\texttt{sub}]\!])\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [v_3, v_2]))) \\
= (\mathit{top}\ (\mathit{arithop}\ (\mathcal{A}\ [\![\texttt{sub}]\!])\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [(\mathit{Int} \mapsto \mathit{Value}\ 5), (\mathit{Int} \mapsto \mathit{Value}\ 3)]))) \\
= (\mathit{top}\ (\mathit{push}\ (\mathcal{A}\ [\![\texttt{sub}]\!]\ (\mathit{Int} \mapsto \mathit{Value}\ 3)\ (\mathit{Int} \mapsto \mathit{Value}\ 5)) \\
\qquad\qquad (\mathit{pop}\ (\mathit{pop}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [(\mathit{Int} \mapsto \mathit{Value}\ 3), (\mathit{Int} \mapsto \mathit{Value}\ 2)]))))) \\
= (\mathit{top}\ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Answer}\ (\mathit{Int} \mapsto \mathit{Value}\ ((3\ -_{\mathrm{Int}}\ 5))))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\,]))) \\
= (\mathit{top}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [(\mathit{Int} \mapsto \mathit{Value}\ \text{-}2)])) \\
= (\mathit{Value} \mapsto \mathit{Answer}\ (\mathit{Int} \mapsto \mathit{Value}\ \text{-}2))
\end{array}
$$

Figure 4.8: Equational proof that the PostFix program `((5 sub) 3 swap exec)` evaluates to -2.

$\mathcal{P}[\![\texttt{((5 sub) 3 swap exec)}]\!]$
$= (top\ (\mathcal{Q}\ [\![\texttt{(5 sub) 3 swap exec}]\!]\ empty\text{-}stack))$
$= (top\ (\mathcal{Q}\ [\![\texttt{3 swap exec}]\!]\ (Value^* \mapsto Stack\ [v_1])))$
where $v_1 = (Stack\text{-}Transform \mapsto Value\ (\mathcal{Q}\ [\![\texttt{(5 sub)}]\!]))$
$= (top\ (\mathcal{Q}\ [\![\texttt{swap exec}]\!]\ (Value^* \mapsto Stack\ [v_2, v_1])))$
where $v_2 = (Int \mapsto Value\ (\mathcal{N}\ [\![3]\!])) = (Int \mapsto Value\ 3)$
$= (top\ (\mathcal{Q}\ [\![\texttt{exec}]\!]\ (Value^* \mapsto Stack\ [v_1, v_2])))$
$= (top\ (\mathcal{Q}\ [\![\texttt{5 sub}]\!]\ (Value^* \mapsto Stack\ [v_2])))$
$= (top\ (\mathcal{Q}\ [\![\texttt{sub}]\!]\ (Value^* \mapsto Stack\ [v_3, v_2])))$
where $v_3 = (Int \mapsto Value\ (\mathcal{N}\ [\![5]\!])) = (Int \mapsto Value\ 5)$
$= (top\ (\mathcal{Q}\ [\![]\!]\ (Value^* \mapsto Stack\ [(Int \mapsto Value\ -2)])))$
$= (Value \mapsto Answer\ (Int \mapsto Value\ -2))$

Figure 4.9: An alternate equational proof that has the flavor of an operational execution.

$$(\mathcal{C}\ [\![\texttt{exec}]\!]) \circ (\mathcal{C}\ [\![\texttt{swap}]\!]) \circ (\mathcal{C}\ [\![3]\!]) \circ (\mathcal{C}\ [\![\texttt{(5 sub)}]\!])$$
$$= \lambda s\ .\ (Value^* \mapsto Stack\ [(Int \mapsto Value\ -2)])$$

This says that the command sequence $[\texttt{(5 sub)}, 3, \texttt{swap}, \texttt{exec}]$ denotes a transform that pushes the integer value -2 on a given stack. Of particular interest is the next to last line of the proof in Figure 4.10, which makes it clear that the program will evaluate to -2 regardless of the contents of the initial stack.

▷ **Exercise 4.1** Use the PostFix denotational semantics to determine the values of the PostFix programs in Exercise 1.1. ◁

$\mathcal{P}[\![\texttt{((5 sub) 3 swap exec)}]\!]$
$= (top\ (\mathcal{Q}\ [\![\texttt{(5 sub) 3 swap exec}]\!]\ empty\text{-}stack))$
$= (top\ (((\mathcal{C}\ [\![\texttt{exec}]\!]) \circ (\mathcal{C}\ [\![\texttt{swap}]\!]) \circ (\mathcal{C}\ [\![3]\!]) \circ (\mathcal{C}\ [\![\texttt{(5 sub)}]\!]))\ empty\text{-}stack))$
$= (top\ ((\lambda s\ .\ (Value^* \mapsto Stack\ [(Int \mapsto Value\ \text{-}2)]))\ empty\text{-}stack))$
$= ((top \circ (\lambda s\ .\ (Value^* \mapsto Stack\ [(Int \mapsto Value\ \text{-}2)])))\ empty\text{-}stack)$
$= ((\lambda s\ .\ (Value \mapsto Answer\ (Int \mapsto Value\ \text{-}2)))\ empty\text{-}stack)$
$= (Value \mapsto Answer\ (Int \mapsto Value\ \text{-}2))$

Figure 4.10: An equational proof that emphasizes the compositional nature of commands.

▷ **Exercise 4.2** Modify the PostFix denotational semantics to handle PostFix2. Include valuation functions for `:`, `(skip)`, and `(exec)`. ◁

▷ **Exercise 4.3** For each of the following, modify the PostFix denotational semantics to handle the specified extensions:

a. The `pair`, `left`, and `right` commands from Exercise 3.24.

b. The `for` and `repeat` commands from Exercise 3.26.

c. The I, `def`, and `get` commands from Exercise 3.27.

◁

▷ **Exercise 4.4** Use the PostFix denotational semantics to prove the following equivalence:

$$(\mathcal{Q}\ [\![C_1\ \ C_2\ \ \ldots C_n]\!]) = (\mathcal{C}\ [\![C_n]\!]) \circ \ldots \circ (\mathcal{C}\ [\![C_2]\!]) \circ (\mathcal{C}\ [\![C_1]\!])$$

◁

▷ **Exercise 4.5** Develop a denotational semantics for the EL language introduced in Section 2.3. ◁

4.3.2 Proving Observational Equivalence Denotationally

Because denotational semantics is compositional, it is a natural tool for proving that it is safe to replace one phrase by another. Recall the following three facts from the operational semantics of PostFix:

1. Two PostFix command sequences are observationally equivalent if they behave indistinguishably in all program contexts.

2. Two PostFix command sequences are transform equivalent if they map equivalent stacks to equivalent stacks.

3. Transform equivalence implies observational equivalence.

Since the PostFix denotational semantics models command sequences as stack transforms, the denotational equivalence of PostFix command sequences corresponds to transform equivalence in the observational framework. So we expect the following theorem:

PostFix Denotational Equivalence Theorem: $(\mathcal{Q}\ [\![Q_1]\!]) = (\mathcal{Q}\ [\![Q_2]\!])$ implies $Q_1 =_{\text{obs}} Q_2$.

Proof: The proof is the same as that for the PostFix Transform Equivalence Theorem where every instance of $Q \sim_Q Q'$ is replace by $(\mathcal{Q}\ [\![Q]\!]) = (\mathcal{Q}\ [\![Q']\!])$. ◇

We can use this theorem to help us prove the behavioral equivalence of two command sequences. For instance, consider our favorite pair of command sequences: $[\texttt{1}, \texttt{add}, \texttt{2}, \texttt{add}]$ and $[\texttt{3}, \texttt{add}]$. Figure 4.11 shows that these are denotationally equivalent, so by the above theorem, they must be observationally equivalent. The equational reasoning in Figure 4.11 uses the following two equalities, whose proofs are left as exercises:

$$(\mathit{with\text{-}integer\&stack}\ f) \circ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Answer}\ (\mathit{Int} \mapsto \mathit{Value}\ i))) = (f\ \ i) \quad (4.1)$$

$$t \circ (\mathit{with\text{-}integer\&stack}\ f) = (\mathit{with\text{-}integer\&stack}\ (\lambda i\ .\ (t \circ (f\ \ i)))) \quad (4.2)$$

where t maps *error-stack* to *error-stack*

It is worth noting that the denotational proof that $[\texttt{1}, \texttt{add}, \texttt{2}, \texttt{add}] =_{\text{obs}} [\texttt{3}, \texttt{add}]$ has a very different flavor than the operational proof of this fact given in the previous chapter. The operational proof worked by case analysis on the initial stack. The denotation proof works purely by equational reasoning – there is no hint of case analysis here.

We have demonstrated that denotational semantics is helpful for proving observational equivalence. But what about termination? Can the denotational definition for PostFix help us prove that all PostFix programs terminate? Unfortunately, the answer is no. We will discuss why shortly when we reconsider the `dup` command from a denotational perspective.

▷ **Exercise 4.6**

a. Prove equation 4.1.

b. Prove equation 4.2.

c. Equation 4.2 requires that t maps *error-stack* to *error-stack*. Show that the equation is not true if this requirement is violated.

◁

▷ **Exercise 4.7** Use the PostFix denotational semantics to either prove or disprove the purported observational equivalences in Exercise 3.20. ◁

$$
\begin{aligned}
&(\mathcal{Q}\ [\![\texttt{1 add 2 add}]\!])\\
&= (\mathcal{C}\ [\![\texttt{add}]\!]) \circ (\mathcal{C}\ [\![\texttt{2}]\!]) \circ (\mathcal{C}\ [\![\texttt{add}]\!]) \circ (\mathcal{C}\ [\![\texttt{1}]\!])\\
&= (\textit{with-integer\&stack}\\
&\qquad (\lambda i_1{}' \,.\, (\textit{with-integer\&stack}\\
&\qquad\qquad\qquad (\lambda i_2{}' \,.\, (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (i_2{}' + i_1{}'))))))))\\
&\quad \circ\ (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (\mathcal{N}\ [\![\texttt{2}]\!]))))\\
&\quad \circ\ (\textit{with-integer\&stack}\\
&\qquad\quad (\lambda i_1 \,.\, (\textit{with-integer\&stack}\\
&\qquad\qquad\qquad (\lambda i_2 \,.\, (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (i_2 + i_1))))))))\\
&\quad \circ\ (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (\mathcal{N}\ [\![\texttt{1}]\!]))))\\
&= (\textit{with-integer\&stack}\\
&\qquad (\lambda i_2{}' \,.\, (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (i_2{}' + 2))))))\\
&\quad \circ\ \ (\textit{with-integer\&stack}\\
&\qquad\quad (\lambda i_2 \,.\, (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (i_2 + 1))))))\\
&= (\textit{with-integer\&stack}\\
&\qquad (\lambda i_3 \,.\ \ (\textit{with-integer\&stack}\\
&\qquad\qquad\qquad (\lambda i_2{}' \,.\ \ (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (i_2{}' + 2))))))\\
&\qquad\qquad \circ\ (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (i_3 + 1))))))\\
&= (\textit{with-integer\&stack}\\
&\qquad (\lambda i_3 \,.\, (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ ((i_3 + 1) + 2))))))\\
&= (\textit{with-integer\&stack}\\
&\qquad (\lambda i_3 \,.\, (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (i_3 + 3))))))\\
&= (\textit{with-integer\&stack}\\
&\qquad (\lambda i_4 \,.\, (\textit{with-integer\&stack}\\
&\qquad\qquad\qquad (\lambda i_3 \,.\, (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (i_3 + i_4))))))))\\
&\quad \circ\ (\textit{push}\ (\textit{Value}\mapsto\textit{Answer}\ (\textit{Int}\mapsto\textit{Value}\ (\mathcal{N}\ [\![\texttt{3}]\!]))))\\
&= (\mathcal{C}\ [\![\texttt{add}]\!]) \circ (\mathcal{C}\ [\![\texttt{3}]\!])\\
&= (\mathcal{Q}\ [\![\texttt{3 add}]\!])
\end{aligned}
$$

Figure 4.11: Proof that $[\texttt{1}, \texttt{add}, \texttt{2}, \texttt{add}]$ and $[\texttt{3}, \texttt{add}]$ are denotationally equivalent. This implies that the two sequences are observationally equivalent.

4.4 Discussion

4.4.1 Technical Difficulties

We hinted above that the denotational definition of PostFix depends crucially on some subtle details. As a hint of the subtlety, consider what happens to our denotational definition if we extend PostFix with our old friend `dup`. A valuation clause for `dup` seems straightforward:

$$\mathcal{C}[\![\texttt{dup}]\!] = \lambda s \,.\, (push\ \ (top\ \ s)\ \ s)$$

At the same time we know that adding `dup` to the language introduces the possibility that programs may not terminate. Yet, the signature for $\mathcal{P}$ declares that programs map to the *Answer* domain, and the *Answer* domain does not include any entity that represents nontermination. What's going on here?

The source of the problem is the recursive structure of the semantic domains for PostFix. As the domain equations show, the *Stack-Transform*, *Stack*, and *Value* domains are mutually recursive:

$$\begin{array}{rcl} \textit{Stack-Transform} & = & \textit{Stack} \to \textit{Stack} \\ \textit{Stack} & = & \textit{Value}^* + \textit{Error} \\ \textit{Value} & = & \textit{Int} + \textit{Stack-Transform} \end{array}$$

These equations imply that a stack may contain as one of its values a function that maps stacks to stacks. But we know from set theory that descriptions of sets that contain themselves (even indirectly) as members are not necessarily well-defined.

In fact, a simple counting argument shows that these equations are nonsensical if interpreted in the normal set-theoretic way. Unwinding the definition for *Stack-Transform* yields:

$$\begin{array}{ll} \multicolumn{2}{l}{\textit{Stack-Transform}} \\ = & \textit{Stack} \to \textit{Stack} \\ = & (\textit{Value}^* + \textit{Error}) \to (\textit{Value}^* + \textit{Error}) \\ = & ((\textit{Int} + \textit{Stack-Transform})^* + \textit{Error}) \to ((\textit{Int} + \textit{Stack-Transform})^* + \textit{Error}) \end{array}$$

The set of functions *Stack-Transform* $\to$ *Stack-Transform* can easily be embedded in the right hand side, so the right hand side is at least as big as the set *Stack-Transform* $\to$ *Stack-Transform*. But for any set A with more than one element, the set of functions $A \to A$ is bigger than A. Even if A is infinite, the set of functions $A \to A$ is a "bigger" infinity! Since the equation requires the right hand side to be a larger set than the left hand side, it has no solutions in the usual theory of sets.

By a clever construction due to Dana Scott, it is possible to show that recursive domain equations that are suitably constrained have a sensible interpretation. It turns out that this construction can introduce nontermination even when it is not explicitly represented in the individual domains. We will say more about this construction in Chapter 7.

4.4.2 Operational vs. Denotational: A Comparison

We have noted in this section that a denotational semantics expresses the meaning of a program in a much more direct way than an operational semantics. Furthermore, the compositional nature of a denotational semantics is a real boon for proving properties of programs and languages. Why would we ever want to choose an operational semantics over a denotational semantics?

For one thing, an operational semantics is usually a more natural medium for expressing the step-by-step nature of program execution. The notion of "step" is an important one: it is at the heart of a mechanistic view of computation; it provides a measure by which computations can be compared (e.g., which takes the fewest steps); and it provides a natural way to talk about nondeterminism (choice between steps) and concurrency (interleaving the steps of more than one process). What counts as a natural step for a program is explicit in the rewrite rules of an SOS. These notions cannot always be expressed straightforwardly in a denotational approach. Furthermore, in computer science, the bottom line is often what actually runs on a machine, and the operational approach is much closer to this bottom line.

From a mathematical perspective, the advantage of an operational semantics is that it's often much easier to construct than a denotational semantics. Since the objects manipulated by an SOS are simple syntactic entities, there are very few constraints on the form of an operational semantics. Any SOS with a deterministic set of rewrite rules specifies a well-defined behavior function from programs to answer expressions. Creating or extending a set of rewrite rules is fairly painless since it rarely requires any deep mathematical reasoning. Of course, the same emphasis on syntax that facilitates the construction of an operational semantics limits its usefulness for reasoning about programs. For example, its difficult to see how some local change to the rewrite rules affects the global properties of a language.

Constructing a denotational semantics, on the other hand, is mathematically much more intensive. It is necessary to build consistent mathematical representations for each kind of meaning object. The difficulty of building such models in general is illustrated by the fact that there was no mathematically viable interpretation for recursive domain equations until Dana Scott invented one in the early 1970s. Since then, a variety of tools and techniques have been developed that make it easier to construct a denotational semantics that maps programs into a restricted set of meanings. Extending this set of meanings requires potentially difficult proofs that the extensions are sound, so most semanticists are content to stick with the well-understood meanings. This class of meanings is large enough, however, to facilitate a wide range of formal reasoning about programs and programming languages.

4.4.3 Denotational Semantics is not a Program

You may notice that the denotational definition for PostFix looks an awful lot like a program. In fact, as we shall see in the next chapter, it is often straightforward to write executable code that mirrors the structure of valuation clauses. Of course, a program has to be explicit about many of the details suppressed in the denotational definition (parsing the concrete syntax, choosing appropriate data structures to represent domain elements, etc.). Furthermore, details of the implementation language may complicate matters. In particular, the correspondence will be much less direct if the implementation programming language does not support first-class procedures.

Although a denotational definition may suggest a program, it can be dangerous to think of it *as* a program. Given that both denotational semantics and SCHEME+ use a notation based on lambda calculus, it is easy to slip into a mindset in which denotational expressions are evaluated just like SCHEME+ expressions. Although this often produces correct results, it won't work in general because the details of SCHEME+ (applicative order evaluation of arguments , no default currying, dynamic typing) are different from those of the mathematical metalanguage (no inherent evaluation order, default currying, typed arguments). More important, program-like simulations are usually operational in nature; this stands in stark contrast with the denotational view that the metalanguage expressions are simply a notation for referring to elements of a functional domain. A valuation function does not describe a process for evaluating an expression. Rather, it is simply a declarative description for a function (i.e., a triple of a source, a target, and a graph). Programming language procedures imply computation; denotational specifications do not.

Despite the above warning, sometimes it *is* useful to think of denotational descriptions as programs, if only for building intuitions about what they mean. This situation is reminiscent of the dy/dx notation in calculus, which teachers and textbooks commonly warn should never be viewed as a fraction. And yet, viewing it as a fraction has many advantages for understanding its meaning as well as for remembering formulae (the chain rule in particular). Similarly, viewing denotational definitions as programs can be helpful, especially for a beginner. However, be sure that you view the lambda notation of the metalanguage as a typed, curried, normal order programming language. And do keep in mind that a function is mathematically quite a different beast from a program.

Reading

Denotational semantics was invented by Christopher Strachey and Dana Scott. For a tutorial introduction to denotational semantics, we recommend the articles by [Ten76] and [Mos90]. [Sto85], [Gor79], and [Sch86] are full length books devoted to denotational semantics.