Chapter 4

Denotational Semantics

But this denoted a foregone conclusion

— *Othello, William Shakespeare*

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, we had to extend the operational semantics with elaborate notions 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 order may be impossible to detect in practice.

An alternative 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]

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

Here, *errorStack* 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 [1, add, 2, add] as well as the sequence [3, 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 [2, mul] is modeled by the *mul2* function, whose graph is sketched below:

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

Then the stack transform of [3, add, 2, mul] = [3, add] @ [2, mul] is simply the function *mul2* ∘ *add3*, whose graph is:

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

[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.

Similarly the stack transform of $[2, \texttt{mul}, 3, \texttt{add}] = [2, \texttt{mul}]$ @ $[3, \texttt{add}]$ is the function *add3* ∘ *mul2*, whose graph is:

$$\{\langle errorStack, errorStack\rangle,\ \langle[\,], errorStack\rangle, \ldots,$$
$$\langle[-1], [1]\rangle,\ \langle[0], [3]\rangle,\ \langle[1], [5]\rangle, \ldots,$$
$$\langle[add3], errorStack\rangle,\ \langle[mul2], errorStack\rangle, \ldots,$$
$$\langle[5,\ 23], [13,\ 23]\rangle,\ \langle[5,\ mul2,\ 17,\ add3], [13,\ mul2,\ 17,\ add3]\rangle, \ldots\}.$$

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 name "denotational semantics" is derived from its focus on the mathematical values that phrases "denote."

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.

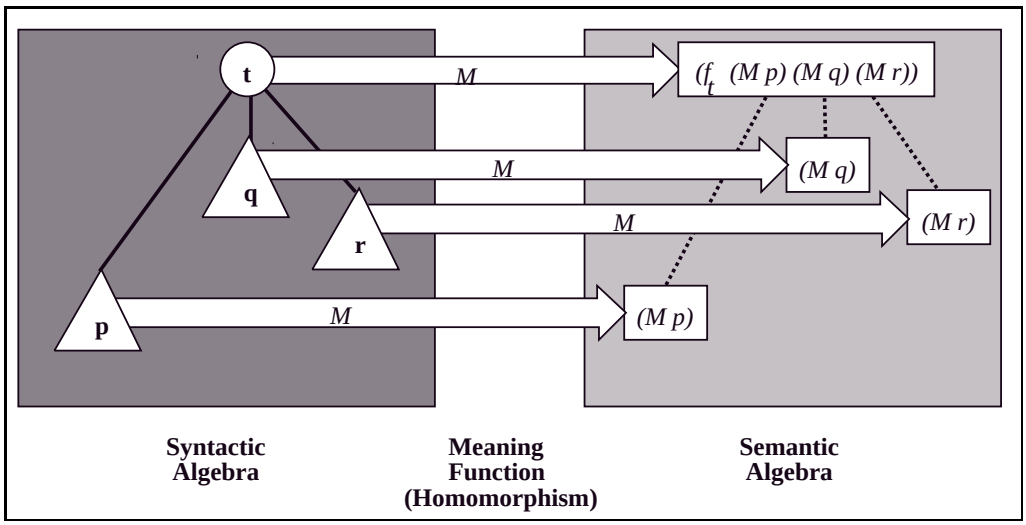

Figure 4.1: The denotational semantics "game board."

Not any function can serve as a meaning function; the function must be a **homomorphism** between the syntactic algebra and the semantic algebra. 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, whose essence is summarized by the motto "the meaning of the whole is composed out of the meaning of the parts".

4.2 A Denotational Semantics for EL

As our first example, we will develop a denotational semantics for the EL expression language. We begin with a pared-down version of the language and show how the semantics changes when adding features to yield full EL.

4.2.1 Step 1: Restricted ELMM

Recall that ELMM (Figure 3.6) is a simple expression language in which programs are expressions, and expressions are trees of binary operators (`+`,`-`,`*`,`/`,`%`) whose leaves are integer numerals. For the moment, let's ignore the `/` and `%` operations, because removing the possibility of divide-by-zero and remainder-by-zero errors simplifies the semantics. In a version of ELMM without `/` and `%`, the meaning of each numeral, expression, and program is just an integer.

This meaning is formalized in Figure 4.2. There is only one semantic domain: the domain *Int* of integers. The meaning of an ELMM program is specified by 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, $\mathcal{P}$ is the valuation function for $P \in$ Program, $\mathcal{NE}$ is the valuation function for $NE \in$ NumExp, and so on.

The meaning $\mathcal{P}[\![\texttt{(elmm } NE_{body}\texttt{)}]\!]$ of an ELMM program `(elmm` NE_{body}`)` is simply the integer $\mathcal{NE}[\![NE_{body}]\!]$ denoted by its body expression NE_{body}. Since an ELMM numerical expression may be either an integer numeral or an arithmetic operation, the definition of $\mathcal{NE}$ has a clause for each of these two cases. In the integer numeral case, the $\mathcal{N}$ function maps the syntactic representation of an integer numeral into a mathematical integer. We will treat integer numerals as atomic entities, but their meaning could be determined in a denotational fashion from their component signs and digits (see Exercise 4.1). In the arithmetic operation case, the $\mathcal{A}$ function maps the operator (one of `+`, `-`, and `*`) into a binary integer function that determines the meaning of the operation from the meanings of the operands.

Figure 4.3 illustrates how the denotational semantics for the restricted version of ELMM can be used to determine the meaning of the sample ELMM program `(elmm (* (+ 1 2) (- 9 5)))`. Because $\mathcal{P}$ maps programs to their meanings, $\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$ *is* the meaning of this program. However, this fact is not very useful as stated because the element of *Int* denoted by the program is not immediately apparent from the form of the metalanguage expression $\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$. 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 such manipulations as:

Semantic Domain

$i \in Int = \{\ldots\ ,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots\}$

Valuation Functions

$\mathcal{P}$: Program $\rightarrow$ *Int*
$\mathcal{P}[\![\texttt{(elmm}\ NE_{body}\texttt{)}]\!] = \mathcal{NE}[\![NE_{body}]\!]$

$\mathcal{NE}$: NumExp $\rightarrow$ *Int*
$\mathcal{NE}[\![N]\!] = (\mathcal{N}[\![N]\!])$
$\mathcal{NE}[\![\texttt{(}A\ \ NE_1\ \ NE_2\texttt{)}]\!] = (\mathcal{A}[\![A]\!]\ \ (\mathcal{NE}[\![NE_1]\!])\ \ (\mathcal{NE}[\![NE_2]\!]))$

$\mathcal{A}$: ArithmeticOperator $\rightarrow$ (*Int* $\rightarrow$ *Int* $\rightarrow$ *Int*)
$\mathcal{A}[\![\texttt{+}]\!] = +_{Int}$
$\mathcal{A}[\![\texttt{-}]\!] = -_{Int}$
$\mathcal{A}[\![\texttt{*}]\!] = \times_{Int}$

$\mathcal{N}$: Intlit $\rightarrow$ *Int*
$\mathcal{N}$ maps integer numerals to the integer numbers they denote.

Figure 4.2: Denotational semantics for a version of ELMM without `/` and `%`.

$\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$
$= \mathcal{NE}[\![\texttt{(* (+ 1 2) (- 9 5))}]\!]$
$= (\mathcal{A}[\![\texttt{*}]\!]\ \ (\mathcal{NE}[\![\texttt{(+ 1 2)}]\!])\ \ (\mathcal{NE}[\![\texttt{(- 9 5)}]\!]))$
$= ((\mathcal{NE}[\![\texttt{(+ 1 2)}]\!]) \times_{Int} (\mathcal{NE}[\![\texttt{(- 9 5)}]\!]))$
$= ((\mathcal{A}[\![\texttt{+}]\!]\ \ (\mathcal{NE}[\![\texttt{1}]\!])\ \ (\mathcal{NE}[\![\texttt{2}]\!])) \times_{Int} (\mathcal{A}[\![\texttt{-}]\!]\ \ (\mathcal{NE}[\![\texttt{9}]\!])\ \ (\mathcal{NE}[\![\texttt{5}]\!])))$
$= ((1 +_{Int} 2) \times_{Int} (9 -_{Int} 5))$
$= (3 \times_{Int} 4)$
$= 12$

Figure 4.3: Meaning of a sample program in restricted ELMM.

- substituting equals for equals;
- applying functions to arguments;
- equating 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 4.3 presents an equational proof that $\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$ is equal to the integer 12. Each equality in the proof is justified by familiar mathematical rules. For example, the equality

$$\mathcal{NE}[\![\texttt{(* (+ 1 2) (- 9 5))}]\!] = (\mathcal{A}[\![\texttt{*}]\!]\ (\mathcal{NE}[\![\texttt{(+ 1 2)}]\!])\ (\mathcal{NE}[\![\texttt{(- 9 5)}]\!]))$$

is justified by the arithmetic operation clause in the definition of $\mathcal{NE}$, while the equality

$$((1 +_{Int} 2) \times_{Int} (9 -_{Int} 5)) = (3 \times_{Int} 4)$$

is justified by algebraic rules for manipulating integers. We emphasize that $\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!]$, as well as every other line in Figure 4.3, denotes exactly the same integer. The whole purpose of the equational proof is to simplify the original expression into another metalanguage expression whose form more directly expresses the meaning of the program.

4.2.2 Step 2: Full ELMM

What happens to the denotational semantics for ELMM if we add back in the `/` and `%` operators? We now have to worry about the meaning of expressions like `(/ 1 0)` and `(% 2 0)`. We will model the meaning of such expressions by the distinguished token *error*. Since ELMM programs, numerical expressions, and arithmetic operators can now return errors in addition to integers, we invent an *Answer* domain with both of these kinds of entities to represent their meanings and change the valuation functions $\mathcal{P}$, $\mathcal{NE}$, and $\mathcal{A}$ accordingly (Figure 4.4). The integer numeral clause for $\mathcal{NE}$ now needs the injection $Int \mapsto Answer$, and the arithmetic operation clause must now propagate any errors found in the operands. The $\mathcal{A}$ clauses for `/` and `%` handle specially the case where the second operand is zero, and $Int \mapsto Answer$ injections must be used in the "regular" cases for all operators.

In full ELMM, the sample program `(elmm (* (+ 1 2) (- 9 5)))` has the meaning $(Int \mapsto Answer\ 12)$. Figure 4.5 presents an equational proof of this fact. All the pattern matching clauses appearing in the proof are there to handle the propagation of errors. The sample program has no errors, but we could

Semantic Domains

$i \in Int = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$Error = \{error\}$
$a \in Answer = Int + Error$

Valuation Functions

$\mathcal{P}$: Program $\rightarrow$ *Answer*

$\mathcal{P}[\![\texttt{(elmm } NE\texttt{)}]\!] = \mathcal{NE}[\![NE]\!]$

$\mathcal{NE}$: NumExp $\rightarrow$ *Answer*

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

$\mathcal{NE}[\![\texttt{(}A\ NE_1\ NE_2\texttt{)}]\!] =$ **matching** $\langle \mathcal{NE}[\![NE_1]\!], \mathcal{NE}[\![NE_2]\!] \rangle$
$\triangleright \langle (Int \mapsto Answer\ i_1), (Int \mapsto Answer\ i_2) \rangle\ [\![\ (\mathcal{A}[\![A]\!]\ i_1\ i_2)$
$\triangleright$ **else** $(Error \mapsto Answer\ error)$ **endmatching**

$\mathcal{A}$: ArithmeticOperator $\rightarrow (Int \rightarrow Int \rightarrow Answer)$

$\mathcal{A}[\![\texttt{+}]\!] = \lambda i_1\, i_2\, .\ (Int \mapsto Answer\ (i_1 +_{Int} i_2))$
`-` and `*` are handled similarly.

$\mathcal{A}[\![\texttt{/}]\!] = \lambda i_1\, i_2\, .$ **if** $i_2 = 0$
then $(Error \mapsto Answer\ error)$
else $(Int \mapsto Answer\ (i_1\ /_{Int}\ i_2))$ **fi**
`%` is handled similarly.

$\mathcal{N}$: Intlit $\rightarrow$ *Int*
$\mathcal{N}$ maps integer numerals to the integer numbers they denote.

Figure 4.4: Denotational semantics for a version of ELMM with `/` and `%`.

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!] \\
= \mathcal{NE}[\![\texttt{(* (+ 1 2) (- 9 5))}]\!] \\
= \textbf{matching}\ \langle \mathcal{NE}[\![\texttt{(+ 1 2)}]\!], \mathcal{NE}[\![\texttt{(- 9 5)}]\!] \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle \textbf{matching}\ \langle \mathcal{NE}[\![\texttt{1}]\!], \mathcal{NE}[\![\texttt{2}]\!] \rangle \\
\qquad\qquad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_3), (\mathit{Int} \mapsto \mathit{Answer}\ i_4) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{+}]\!]\ i_3\ i_4) \\
\qquad\qquad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\,, \\
\qquad\qquad \textbf{matching}\ \langle \mathcal{NE}[\![\texttt{9}]\!], \mathcal{NE}[\![\texttt{5}]\!] \rangle \\
\qquad\qquad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_5), (\mathit{Int} \mapsto \mathit{Answer}\ i_6) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{-}]\!]\ i_5\ i_6) \\
\qquad\qquad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\ \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle \textbf{matching}\ \langle (\mathit{Int} \mapsto \mathit{Answer}\ 1), (\mathit{Int} \mapsto \mathit{Answer}\ 2) \rangle \\
\qquad\qquad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_3), (\mathit{Int} \mapsto \mathit{Answer}\ i_4) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{+}]\!]\ i_3\ i_4) \\
\qquad\qquad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\,, \\
\qquad\qquad \textbf{matching}\ \langle (\mathit{Int} \mapsto \mathit{Answer}\ 9), (\mathit{Int} \mapsto \mathit{Answer}\ 5) \rangle \\
\qquad\qquad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_5), (\mathit{Int} \mapsto \mathit{Answer}\ i_6) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{-}]\!]\ i_5\ i_6) \\
\qquad\qquad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\ \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle (\mathcal{A}[\![\texttt{+}]\!]\ 1\ 2)\,, (\mathcal{A}[\![\texttt{-}]\!]\ 9\ 5) \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle (\mathit{Int} \mapsto \mathit{Answer}\ (1 +_{\mathit{Int}} 2)), (\mathit{Int} \mapsto \mathit{Answer}\ (9 +_{\mathit{Int}} 5)) \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= \textbf{matching}\ \langle (\mathit{Int} \mapsto \mathit{Answer}\ 3), (\mathit{Int} \mapsto \mathit{Answer}\ 4) \rangle \\
\quad \triangleright\, \langle (\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2) \rangle\ [\!]\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
\quad \triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
= (\mathcal{A}[\![\texttt{*}]\!]\ 3\ 4) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ (3 \times_{\mathit{Int}} 4)) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ 12)
\end{array}
$$

Figure 4.5: Meaning of a sample program in full ELMM.

introduce one by replacing the subexpression `(- 9 5)` by `(/ 9 0)`. Then the part of the proof beginning

$$= \textbf{matching}\ \langle(\mathcal{A}[\![\texttt{+}]\!]\ 1\ 2), (\mathcal{A}[\![\texttt{-}]\!]\ 9\ 5)\rangle\ \ldots$$

would become:

$$\begin{aligned}
&= \textbf{matching}\ \langle(\mathcal{A}[\![\texttt{+}]\!]\ 1\ 2), (\mathcal{A}[\![\texttt{/}]\!]\ 9\ 0)\rangle \\
&\quad \triangleright \langle(\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2)\rangle\ [\![\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
&\quad \triangleright \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
&= \textbf{matching}\ \langle(\mathit{Int} \mapsto \mathit{Answer}\ (1 +_{\mathit{Int}} 2)), (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\rangle \\
&\quad \triangleright \langle(\mathit{Int} \mapsto \mathit{Answer}\ i_1), (\mathit{Int} \mapsto \mathit{Answer}\ i_2)\rangle\ [\![\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2) \\
&\quad \triangleright \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching} \\
&= (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error}).
\end{aligned}$$

Expressing error propagation via explicit pattern matching makes the equational proof in Figure 4.5 rather messy. As in programming, in denotational semantics it is good practice to create abstractions that capture common patterns of behavior and hide messy details. This can improve the clarity of the definitions and proofs while at the same time making them more compact.

We illustrate this kind of abstraction by introducing the following higher-order function for simplifying error handling in ELMM:

$$\begin{aligned}
&\mathit{with\text{-}int} : \mathit{Answer} \to (\mathit{Int} \to \mathit{Answer}) \to \mathit{Answer} \\
&= \lambda a f\ .\ \textbf{matching}\ \ a \\
&\qquad \triangleright (\mathit{Int} \mapsto \mathit{Answer}\ i)\ [\![\ (f\ \ i) \\
&\qquad \triangleright \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}\ .
\end{aligned}$$

with-int takes an answer a and a function f from integers to answers and returns an answer. It automatically propagates errors, in the sense that it maps an input error answer to an output error answer. The function f specifies what is done for inputs that are integer answers. Thus, *with-int* hides details of error handling and extracting integers from integer answers.

A metalanguage expression of the form $(\mathit{with\text{-}int}\ \ a\ \ (\lambda i\ .\ E))$ serves as a kind of binding construct, i.e., a construct that introduces a name for a value. One way to pronounce this is:

> "If a is an integer answer, then let i name the integer in E and return the value of E. Otherwise, a must be an error, in which case an error should be returned."

The following equalities involving *with-int* are useful:

$$(\mathit{with\text{-}int}\ \ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ f) = (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})$$

$$(\mathit{with\text{-}int}\ \ (\mathit{Int} \mapsto \mathit{Answer}\ i)\ f) = (f\ \ i)$$

$$(\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![N]\!])\ \ f) = (f\ \ (\mathcal{N}[\![N]\!]))$$

$$
\begin{aligned}
&\mathcal{P}[\![\texttt{(elmm (* (+ 1 2) (- 9 5)))}]\!] \\
&= \mathcal{NE}[\![\texttt{(* (+ 1 2) (- 9 5))}]\!] \\
&= \textit{with-int}\ (\mathcal{NE}[\![\texttt{(+ 1 2)}]\!]) \\
&\quad (\lambda i_1\ .\ \textit{with-int}\ (\mathcal{NE}[\![\texttt{(- 9 5)}]\!]) \\
&\qquad (\lambda i_2\ .\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2))) \\
&= \textit{with-int}\ (\textit{with-int}\ (\mathcal{NE}[\![\texttt{1}]\!]) \\
&\qquad (\lambda i_3\ .\ \textit{with-int}\ (\mathcal{NE}[\![\texttt{2}]\!]) \\
&\qquad\quad (\lambda i_4\ .\ (\mathcal{A}[\![\texttt{+}]\!]\ i_3\ i_4)))) \\
&\quad (\lambda i_1\ .\ \textit{with-int}\ (\textit{with-int}\ (\mathcal{NE}[\![\texttt{9}]\!]) \\
&\qquad\quad (\lambda i_5\ .\ \textit{with-int}\ (\mathcal{NE}[\![\texttt{5}]\!]) \\
&\qquad\qquad (\lambda i_6\ .\ (\mathcal{A}[\![\texttt{-}]\!]\ i_5\ i_6)))) \\
&\qquad (\lambda i_2\ .\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2))) \\
&= \textit{with-int}\ (\mathcal{A}[\![\texttt{+}]\!]\ 1\ 2) \\
&\quad (\lambda i_1\ .\ \textit{with-int}\ (\mathcal{A}[\![\texttt{-}]\!]\ 9\ 5) \\
&\qquad (\lambda i_2\ .\ (\mathcal{A}[\![\texttt{*}]\!]\ i_1\ i_2))) \\
&= \textit{with-int}\ (\textit{Int} \mapsto \textit{Answer}\ (1 +_{\textit{Int}} 2)) \\
&\quad (\lambda i_1\ .\ \textit{with-int}\ (\textit{Int} \mapsto \textit{Answer}\ (9 -_{\textit{Int}} 5)) \\
&\qquad (\lambda i_2\ .\ (\textit{Int} \mapsto \textit{Answer}\ (i_1 \times_{\textit{Int}} i_2)))) \\
&= (\textit{Int} \mapsto \textit{Answer}\ ((1 +_{\textit{Int}} 2) \times_{\textit{Int}} (9 -_{\textit{Int}} 5))) \\
&= (\textit{Int} \mapsto \textit{Answer}\ (3 \times_{\textit{Int}} 4)) \\
&= (\textit{Int} \mapsto \textit{Answer}\ 12)
\end{aligned}
$$

Figure 4.6: Example illustrating how *with-int* hides error propagation.

Using *with-int*, the $\mathcal{NE}$ valuation clause for arithmetic expressions can be redefined as:

$$
\begin{aligned}
&\mathcal{NE}[\![\texttt{(}A\ \ NE_1\ \ NE_2\texttt{)}]\!] \\
&= \textit{with-int}\ \ (\mathcal{NE}[\![NE_1]\!])\ \ (\lambda i_1\ .\ (\textit{with-int}\ \ (\mathcal{NE}[\![NE_2]\!])\ \ (\lambda i_2\ .\ (\mathcal{A}[\![A]\!]\ \ i_1\ \ i_2)))).
\end{aligned}
$$

With this modified definition and the above *with-int* equalities, details of error propagation can be hidden in equational proofs for ELMM meanings (see Figure 4.6).

One of the powers of lambda notation is that it supports the invention of new binding constructs like *with-int* 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 practical programming under as **monadic style** (Chapter 8) and **continuation passing style** (Chapters 9 and 17).

4.2.3 Step 3: ELM

The ELM language (Exercise 3.10) is obtained from ELMM by adding indexed input via the expression `(arg` N_{index}`)`, where N_{index} specifies the index (starting at 1) of a program argument. The program form is `(elm` $N_{numargs}$ NE_{body}`)`, where N_{numarg} indicates the number of integer arguments expected by the program when it is executed. Intuitively, the meaning of ELM programs and numerical expressions must now be extended to include the program arguments. In Figure 4.7, this is expressed by modeling the meaning of programs and expressions as functions with signature $Int^* \rightarrow Answer$ that map a sequence of integers (the program arguments) to an answer (either an integer or an error). The program argument sequence i^* must be "passed down" the syntax tree to the body of a program and the operands of an arithmetic operation so that they can eventually be referenced in an `arg` form at a leaf of the syntax tree. The `elm` program form must check that the number of supplied arguments matches the expected number of arguments $i_{numargs}$, and the `arg` form must check that the index i_{index} is between 1 and the number of arguments, inclusive.

Figure 4.8 uses denotational definitions to find the result of applying the ELM program `(elm 2 (+ (arg 2) (* (arg 1) 3)))` to the argument sequence $[4, 5]$. The equational proof assumes the following equalities, which are easy to verify:

$$(\textit{with-int}\ \ (\mathcal{NE}[\![N]\!]\ \ i^*)\ \ f) = (f\ \ (\mathcal{N}[\![N]\!]))$$

$$(\textit{with-int}\ \ (\mathcal{NE}[\![\texttt{(arg}\ N\texttt{)}]\!]\ \ [i_1, \ldots, i_k, \ldots, i_n])\ \ f) = (f\ \ i_k)\text{, where } \mathcal{N}[\![N]\!] = \mathrm{k}$$

$$(\textit{with-int}\ \ (\mathcal{A}[\![A]\!]\ \ i_1\ \ i_2)\ \ f) = (f\ \ i_{res})\text{, where } (\mathcal{A}[\![A]\!]\ \ i_1\ \ i_2) = (\textit{Int} \mapsto \textit{Result}\ \ i_{res})$$

In Figure 4.8, if we replace the concrete argument integers 4 and 5 by abstract integers i_{arg1} and i_{arg2}, respectively, then the result would be

$$(\textit{Int} \mapsto \textit{Answer}\ \ (i_{arg2} +_{\textit{Int}}\ (i_{arg1} \times_{\textit{Int}} 3))).$$

Based on this observation, we can give a meaning to the sample program itself (i.e., without applying it to particular arguments). Such a meaning must be abstracted over an arbitrary argument sequence:

$$\begin{aligned}
&\mathcal{P}[\![\texttt{(elmm 2 (+ (arg 2) (* (arg 1) 3)))}]\!] \\
&= \lambda i^*.\ \textbf{matching}\ \ i^* \\
&\qquad \triangleright\, [i_{arg1}, i_{arg2}]\ [\![\ (\textit{Int} \mapsto \textit{Answer}\ \ (i_{arg2} +_{\textit{Int}}\ (i_{arg1} \times_{\textit{Int}} 3))) \\
&\qquad \triangleright\, \textbf{else}\ (\textit{Error} \mapsto \textit{Answer}\ \ \textit{error})\ \textbf{endmatching}\,.
\end{aligned}$$

Here we have translated the **if** that appears in the $\mathcal{P}$ definition in Figure 4.7 into an equivalent **matching** construct that gives the names i_{arg1} and i_{arg2} to

Semantic Domains

$i \in \mathit{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathit{Error} = \{\mathit{error}\}$
$a \in \mathit{Answer} = \mathit{Int} + \mathit{Error}$

Valuation Functions

$\mathcal{P} : \text{Program} \to \mathit{Int}^* \to \mathit{Answer}$

$\mathcal{P}[\![(\texttt{elm}\ N_{numargs}\ NE_{body})]\!]$
$= \lambda i^* .\ \textbf{if}\ (\mathit{length}\ i^*) = \mathcal{N}[\![N_{numargs}]\!]$
$\textbf{then}\ \mathcal{NE}[\![NE]\!]\ i^*$
$\textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{fi}$

$\mathcal{NE} : \text{NumExp} \to \mathit{Int}^* \to \mathit{Answer}$

$\mathcal{NE}[\![N_{num}]\!] = \lambda i^* .\ (\mathit{Int} \mapsto \mathit{Answer}\ \mathcal{N}[\![N_{num}]\!])$

$\mathcal{NE}[\![(\texttt{arg}\ N_{index})]\!] = \lambda i^* .\ \textbf{if}\ (1 \leq \mathcal{N}[\![N_{index}]\!])\ \textbf{and}\ (\mathcal{N}[\![N_{index}]\!] \leq (\mathit{length}\ i^*))$
$\textbf{then}\ (\mathit{Int} \mapsto \mathit{Answer}\ (\mathit{nth}\ (\mathcal{N}[\![N_{index}]\!])\ i^*))$
$\textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{fi}$

$\mathcal{NE}[\![(A\ NE_1\ NE_2)]\!]$
$= (\mathit{with\text{-}int}\ (\mathcal{NE}[\![NE_1]\!]\ i^*)\ (\lambda i_1 .\ (\mathit{with\text{-}int}\ (\mathcal{NE}[\![NE_2]\!]\ i^*)\ (\lambda i_2 .\ (\mathcal{A}[\![A]\!]\ i_1\ i_2)))))$

$\mathcal{A}$ and $\mathcal{N}$ are unchanged from ELMM.

Figure 4.7: Denotational semantics for ELM.

$\mathcal{P}[\![$`(elm 2 (+ (arg 2) (* (arg 1) 3)))`$]\!]$ $[4,5]$
$=$ **if** $(length\ [4,5]) = \mathcal{N}[\![$`2`$]\!]$
  then $(\mathcal{NE}[\![$`(+ (arg 2) (* (arg 1) 3))`$]\!]\ [4,5])$
  else $(Error \mapsto Answer\ error)$
$= (\mathcal{NE}[\![$`(+ (arg 2) (* (arg 1) 3))`$]\!]\ [4,5])$
$=$ *with-int* $(\mathcal{NE}[\![$`(arg 2)`$]\!]\ [4,5])$
  $(\lambda i_1\ .$ *with-int* $(\mathcal{NE}[\![$`(* (arg 1) 3)`$]\!]\ [4,5])$
    $(\lambda i_2\ .\ (\mathcal{A}[\![$`+`$]\!]\ i_1\ i_2)))$
$=$ *with-int* (*with-int* $(\mathcal{NE}[\![$`(arg 1)`$]\!]\ [4,5])$
    $(\lambda i_3\ .$ *with-int* $(\mathcal{NE}[\![$`3`$]\!]\ [4,5])$
      $(\lambda i_4\ .\ (\mathcal{A}[\![$`*`$]\!]\ i_3\ i_4)))$
  $(\lambda i_2\ .\ (\mathcal{A}[\![$`+`$]\!]\ 5\ i_2))$
$=$ (*with-int* $(\mathcal{A}[\![$`*`$]\!]\ 4\ 3)\ (\lambda i_2\ .\ (\mathcal{A}[\![$`+`$]\!]\ 5\ i_2)))$
$= (\mathcal{A}[\![$`+`$]\!]\ 5\ 12)$
$(Int \mapsto Answer\ 17)$

Figure 4.8: Meaning of an ELM program applied to two arguments.

the two integer arguments in the case where the argument sequence i^* has two elements. We showed above that the result in this case is correct, and we know that an error is returned for any other length.

4.2.4 Step 4: EL

Full EL (Figure 2.4) is obtained from ELM by adding a numerical `if` expression and boolean expressions for controlling these expressions. Boolean expressions BE include the truth literals `true` and `false`, relational expressions like `(<` NE_1 NE_2`)`, and logical expressions like `(and` BE_1 BE_2`)`. Since boolean expressions can include numerical expressions as subexpressions and such subexpressions can denote errors, boolean expressions can also denote errors (e.g. `(< 1 (/ 2 0))`). In Figure 4.9, we model this by having the valuation function $\mathcal{BE}$ for boolean expressions return an element in the domain *BoolAnswer* of "boolean answers" that is distinct from the domain *Answer* of "integer answers". Since a numerical subexpression of a relational expression could be an `arg` expression, the meaning of a boolean expression is a function with signature $Int^* \rightarrow BoolAnswer$ that maps implicit program arguments to a boolean answer. The error handling for relational and logical operations is handled by $\mathcal{BE}$, so the $\mathcal{R}$ and $\mathcal{L}$ valuation functions manipulate only non-error values.

Note that the error-handling in $\mathcal{BE}[\![$`(`R_{rator} NE_1 NE_2`)`$]\!]$ is performed by

Semantic Domains

$i \in Int = \{\ldots\ ,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots\}$
$b \in Bool = \{true, false\}$
$Error = \{error\}$
$a \in Answer = Int\ +\ Error$
$ba \in BoolAnswer = Bool\ +\ Error$

Valuation Functions

$\mathcal{P}$: Program $\rightarrow$ $Int^* \rightarrow Answer$

The $\mathcal{P}$ clause is unchanged from ELM (except the keyword `elm` becomes `el`).

$\mathcal{NE}$: NumExp $\rightarrow$ $Int^* \rightarrow Answer$

$\mathcal{NE}[\![\texttt{(if}\ BE_{test}\ NE_{then}\ NE_{else}\texttt{)}]\!]$
$= \lambda i^*\ .\ \textbf{matching}\ (\mathcal{BE}[\![BE_{test}]\!]\ i^*)$
$\triangleright\ (Bool \mapsto BoolAnswer\ b)\ [\![$
$\textbf{if}\ b\ \textbf{then}\ \mathcal{NE}[\![NE_{then}]\!]\ i^*\ \textbf{else}\ \mathcal{NE}[\![NE_{else}]\!]\ i^*\ \textbf{fi}$
$\triangleright\ \textbf{else}\ (Error \mapsto Answer\ error)\ \textbf{endmatching}$

The other $\mathcal{NE}$ clauses are unchanged from ELM.

$\mathcal{BE}$: BoolExp $\rightarrow$ $Int^* \rightarrow BoolAnswer$

$\mathcal{BE}[\![\texttt{true}]\!] = \lambda i^*\ .\ (Bool \mapsto BoolAnswer\ true)$

$\mathcal{BE}[\![\texttt{false}]\!] = \lambda i^*\ .\ (Bool \mapsto BoolAnswer\ false)$

$\mathcal{BE}[\![\texttt{(}R_{rator}\ NE_1\ NE_2\texttt{)}]\!]$
$= \lambda i^*\ .\ \textbf{matching}\ \langle \mathcal{NE}[\![NE_1]\!]\ i^*, \mathcal{NE}[\![NE_2]\!]\ i^* \rangle$
$\triangleright\ \langle (Int \mapsto Answer\ i_1), (Int \mapsto Answer\ i_2) \rangle\ [\![$
$(Bool \mapsto BoolAnswer\ (\mathcal{R}[\![R]\!]\ i_1\ i_2))$
$\triangleright\ \textbf{else}\ (Error \mapsto BoolAnswer\ error)\ \textbf{endmatching}$

$\mathcal{BE}[\![\texttt{(}L_{rator}\ BE_1\ BE_2\texttt{)}]\!]$
$= \lambda i^*\ .\ \textbf{matching}\ \langle \mathcal{BE}[\![BE_1]\!]\ i^*, \mathcal{BE}[\![BE_2]\!]\ i^* \rangle$
$\triangleright\ \langle (Bool \mapsto BoolAnswer\ b_1), (Bool \mapsto BoolAnswer\ b_2) \rangle\ [\![$
$(Bool \mapsto BoolAnswer\ (\mathcal{L}[\![L]\!]\ b_1\ b_2))$
$\triangleright\ \textbf{else}\ (Error \mapsto BoolAnswer\ error)\ \textbf{endmatching}$

$\mathcal{R}$: RelationalOperator $\rightarrow$ $(Int \rightarrow Int \rightarrow Bool)$

$\mathcal{R}[\![\texttt{<}]\!] = <_{Int}$
`=` and `>` are handled similarly.

$\mathcal{L}$: LogicalOperator $\rightarrow$ $(Bool \rightarrow Bool \rightarrow Bool)$

$\mathcal{L}[\![\texttt{and}]\!] = \lambda b_1\, b_2\ .\ (b_1\ \textbf{and}\ b_2)$
`or` is handled similarly.

$\mathcal{A}$ and $\mathcal{N}$ are unchanged from ELM.

Figure 4.9: Denotational semantics for EL.

pattern matching. Could it instead be done via *with-int*? No. The final return value of *with-int* is in *Answer*, but the final return value of $\mathcal{BE}$ is in *BoolAnswer*. However, we could define and use a new auxiliary function that is like *with-int* but returns an element of *BoolAnswer*(see Exercise 4.3).

Something that stands out in our study of the denotational semantics of the EL dialects is the importance of semantic domains and the signatures of valuation functions. Studying these gives insight into the fundamental nature of a language, even if the detailed valuation clause definitions are unavailable. For example, consider the signature of the numerical expression valuation function $\mathcal{NE}$ in the various dialects we studied. In ELMM without `/` and `%`, the signature

$$\mathcal{NE} : \text{NumExp} \rightarrow \mathit{Int}$$

indicates that expressions simply stands for an integer. In full ELMM, the "unwound" signature

$$\mathcal{NE} : \text{NumExp} \rightarrow (\mathit{Int} + \mathit{Error})$$

indicates that errors may be encountered in the evaluation of some expressions. The ELM signature

$$\mathcal{NE} : \text{NumExp} \rightarrow \mathit{Int}^* \rightarrow (\mathit{Int} + \mathit{Error})$$

has a context domain Int^* representing program arguments that are passed down the abstract syntax tree. We will see many kinds of contexts in our study of other languages. Some, like ELM program arguments, only flow down to subexpressions. We shall see later that other contexts can have more complex flows, and that these flows are reflected in the valuation function signatures.

4.2.5 A Denotational Semantics is Not a Program

You may have noticed that the denotational definitions for the dialects of EL strongly resemble programs in certain programming languages. In fact, it is straightforward to write an executable EL interpreter that reflects the structure of its valuation clauses, especially in functional programming languages like ML, HASKELL, and SCHEME. Of course, an interpreter 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 often suggests an approach for implementing an interpreter program, it can be misleading to think of the denotational definition itself *as* a program. Programming language procedures typically imply computation; denotational specifications do not. An interpreter specifies a process for evaluating program phrases, often one with particular operational properties. In contrast, there is no notion of process associated with a valuation function: it is simply a declarative description for a mathematical function (i.e., a triple of a source, a target, and a graph).

For example, consider the following metalanguage expression, which might arise in the context of reasoning about an ELMM program:

$$\lambda i_0 \,.\, \textit{with-int}\ \ (\mathcal{A}[\![/]\!]\ \ i_0\ \ 2)\ \ (\lambda i_1 \,.\, (\textit{with-int}\ \ (\mathcal{A}[\![-]\!]\ \ 3\ \ 3)\ \ (\lambda i_2 \,.\, (\mathcal{A}[\![*]\!]\ \ i_1\ \ i_2))))\,.$$

If we (incorrectly) view this as an expression in a programming language like ML or SCHEME, we might think that no evaluation can take place until an integer is supplied for i_0, and, after that happens, the division must be performed first, followed by the subtraction, and finally the multiplication. But there is no inherent notion of evaluation order associated with the metalanguage expression. We can perform any mathematical simplifications in any order on this expression. For example, observing that $(\mathcal{A}[\![-]\!]\ 3\ 3)$ has the same meaning as $(\mathcal{N}[\![0]\!])$ allows us to rewrite the expression to

$$\lambda i_0 \,.\, \textit{with-int}\ \ (\mathcal{A}[\![/]\!]\ \ i_0\ \ 2)\ \ (\lambda i_1 \,.\, (\textit{with-int}\ \ (\mathcal{N}[\![0]\!])\ \ (\lambda i_2 \,.\, (\mathcal{A}[\![*]\!]\ \ i_1\ \ i_2))))\,.$$

This is equivalent to

$$\lambda i_0 \,.\, \textit{with-int}\ \ (\mathcal{A}[\![/]\!]\ \ i_0\ \ 2)\ \ (\lambda i_1 \,.\, (\mathcal{A}[\![*]\!]\ \ i_1\ \ 0))\,,$$

which is in turn equivalent to

$$\lambda i_0 \,.\, \textit{with-int}\ \ (\mathcal{A}[\![/]\!]\ \ i_0\ \ 2)\ \ (\lambda i_1 \,.\, (\textit{Int} \mapsto \textit{Answer}\ \ 0))\,,$$

since the product of 0 and any integer is 0. A division result cannot be an error when the second argument is non-zero, so this can be further simplified to:

$$\lambda i_0 \,.\, (\textit{Int} \mapsto \textit{Answer}\ \ 0).$$

The moral of this example is that many simplifications can be done with metalanguage expressions that would be difficult to justify with expressions in most programming languages.[2]

[2]Certain real-world programming languages, particularly the purely functional language HASKELL, were designed to support the kind of mathematical reasoning that can be done with metalanguage expressions.

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 sometimes be helpful, especially for a beginner. To avoid misleading processing intuitions from familiar programming languages, you should view the lambda notation of the metalanguage as a typed, curried, normal-order programming language.

▷ **Exercise 4.1** We have treated integer numerals atomically, but we could express them in terms of their component signs and digits via an s-expression grammar:

SN ∈ SignedNumeral
UN ∈ UnsignedNumeral
D ∈ Digit
SN ::= `(+` *UN*`)` | `(-` *UN*`)` | *UN*
UN ::= *D* | `(@` *UN* *D*`)`
D ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

For example, the numeral traditionally written as `-273` would be written in s-expression form as `(- (@ (@ 2 7) 3))`. Give a denotational semantics for numerals by providing valuation functions for each of SignedNumeral, UnsignedNumeral, and Digit. ◁

▷ **Exercise 4.2** Use the ELM semantics to determine the meaning of the following ELM program: `(elm 2 (/ (arg 1) (- (arg 1) (arg 2))))`. ◁

▷ **Exercise 4.3** o By analogy with the *with-int* auxiliary function in the ELM semantics, define functions with the following signatures and use them to "hide" error-handling in the EL valuation clauses for conditional expressions, relational operations, and logical operations:

$$\textit{with-bool} : \textit{BoolAnswer} \to (\textit{Bool} \to \textit{Answer}) \to \textit{Answer}$$
$$\textit{with-int}_{BA} : \textit{Answer} \to (\textit{Int} \to \textit{BoolAnswer}) \to \textit{BoolAnswer}$$
$$\textit{with-bool}_{BA} : \textit{BoolAnswer} \to (\textit{Bool} \to \textit{BoolAnswer}) \to \textit{BoolAnswer}$$ ◁

4.3 A Denotational Semantics for PostFix

We are now ready to flesh out the details of the denotational description of PostFix that were sketched in Section 4.1. The abstract syntax for PostFix was already provided in Figure 2.8, so the syntactic algebra is already taken care of. We therefore need to construct the semantic algebra and the meaning function.

$$
\begin{array}{l}
t \in \mathit{StackTransform} = \mathit{Stack} \rightarrow \mathit{Stack} \\
s \in \mathit{Stack} = \mathit{Value}^* + \mathit{Error} \\
v \in \mathit{Value} = \mathit{Int} + \mathit{StackTransform} \\
r \in \mathit{Result} = \mathit{Value} + \mathit{Error} \\
a \in \mathit{Answer} = \mathit{Int} + \mathit{Error} \\
\quad \mathit{Error} = \{\mathit{error}\} \\
i \in \mathit{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
b \in \mathit{Bool} = \{\mathit{true}, \mathit{false}\}
\end{array}
$$

Figure 4.10: Semantic domains for the PostFix denotational semantics.

4.3.1 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 modeled 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, *errorStack*, that indicates that an error has occurred. For example, calling the transform associated with the `swap` command on a stack with fewer than two elements should return *errorStack*. All transforms should return *errorStack* when given *errorStack* as an argument.

Figure 4.10 presents domain equations that describe one implementation of this approach. The *StackTransform* 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 modeled 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. The *Result* domain models intermediate results obtained via stack manipulations or arithmetic operations. It includes an error result to model situations like popping an empty stack and dividing by zero. The *Answer* domain models the final outcome of a PostFix program. Like *Result*, *Answer* includes an error answer, but its only non-error answers are integers (because executable sequences at the top of a final stack cannot be observed and are treated as errors).

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 5 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 4.11 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. *errorResult*, *errorAnswer*, *errorStack*, and *errorTransform* are just names for useful constants involving errors. *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 *Result* rather than just *Value* because it must return *errorResult* in the case where the given stack is empty. *push* takes its argument from *Result* rather than *Value* so that it can be composed with *top*. *intAt* is an auxiliary function that simplifies the specification of `nget`. *arithop* simplifies the specifications 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 a result that is expected to be a transform turns out to be an integer or an error result instead. *resToAns* handles the conversion from results to answers.

The signatures of the functions, especially the stack functions, may seem strange at first glance, because few of them explicitly refer to the *Stack* domain. But recall that *StackTransform* is defined to be $Stack \to Stack$, so that the signature of *push*, for instance, is really

$$Result \to (Stack \to Stack)\,.$$

From this perspective, *push* probably seems more familiar: it is a function that takes an result 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 a result and returns a stack transform. Of course, in either case *push* is exactly the same mathematical entity; the only difference is in how we think about it!

- *errorResult* : *Result*
  An error in the domain *Result*.
- *errorAnswer* : *Answer*
  An error in the domain *Answer*.
- *errorStack* : *Stack*
  The distinguished error stack.
- *errorTransform* : *StackTransform*
  A transform that maps all stacks to *errorStack*.
- *push* : *Result* → *StackTransform*
  Given the result value v, return a transform that pushes v on a stack; otherwise return *errorTransform*.
- *pop* : *StackTransform*
  For a nonempty stack s, return the stack resulting from popping the top value; otherwise return *errorStack*.
- *top* : *Stack* → *Result*
  Given a nonempty stack s, return result that is the top element of s; otherwise return *errorResult*.
- *intAt* : *Int* → *Stack* → *Result*
  Given an integer i_{index} and a stack whose i_{index}th element (starting from 1) is the integer i_{result}, return i_{result}; otherwise return *errorResult*.
- *arithop* : (*Int* → *Int* → *Result*) → *StackTransform*
  Let f : *Int* → *Int* → *Result* 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 (*Value* ↦ *Result* v_{result}) is the result of the application (f i_2 i_1). If the given stack is not of this form or if the result of applying f is *errorResult*, then return *errorStack*.
- *transform* : *Result* → *StackTransform*
  Given a result that is a stack transform, return it; otherwise return *errorTransform*.
- *resToAns* : *Result* → *Answer*
  Given a result that is an integer, return it as an answer; otherwise return *errorAnswer*.

Figure 4.11: Specifications for functions on PostFix semantic domains.

$\mathcal{P}$: Program $\rightarrow$ *Int** $\rightarrow$ *Answer*
$\mathcal{Q}$: Commands $\rightarrow$ *StackTransform*
$\mathcal{C}$: Command $\rightarrow$ *StackTransform*
$\mathcal{A}$: ArithmeticOperator $\rightarrow$ (*Int* $\rightarrow$ *Int* $\rightarrow$ *Result*)
$\mathcal{R}$: RelationalOperator $\rightarrow$ (*Int* $\rightarrow$ *Int* $\rightarrow$ *Bool*)
$\mathcal{N}$: Intlit $\rightarrow$ *Int*

Figure 4.12: Signatures of the PostFix valuation functions.

4.3.2 A Meaning Function for PostFix

Now we're ready to study the meaning function for PostFix. As in EL, we specify the meaning function by a collection of valuation functions, one for each syntactic domain defined by the abstract syntax for the language.

As we learned in studying the denotational semantics of EL, 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 4.12. In the case of PostFix, one of the things the signatures say is that a PostFix program is like an EL program: it takes a sequence of integers as arguments and either returns an integer or signals an error. If the signature of $\mathcal{P}$ were instead

$$\mathcal{P} : \text{Program} \rightarrow \mathit{Int}^* \rightarrow \mathit{Result},$$

it would indicate that some PostFix programs could return a stack transform (i.e., an executable sequence) instead of an integer. If the signature were one of

$$\mathcal{P} : \text{Program} \rightarrow \mathit{Int}^* \rightarrow \mathit{Int} \quad \text{or} \quad \mathcal{P} : \text{Program} \rightarrow \mathit{Int}^* \rightarrow \mathit{Value},$$

it would tell use that errors could not be signaled by a PostFix program.

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. The return type of $\mathcal{A}$ matches the argument type of *arithop*, one of the auxiliary functions specified in Figure 4.11. This is more than coincidence; the auxiliary functions and valuation functions were designed to dovetail in a nice way.

Now we're ready to study the definitions of the PostFix valuation functions, which appear in Figure 4.13. The meaning of a program (`postfix` $N_{numargs}$ Q) is a function that transforms an initial stack consisting of the integers in the argument sequence i^* via the transform $\mathcal{Q}[\![Q]\!]$ and returns the top integer of the

$\mathcal{P}[\![(\texttt{postfix}\ N_{numargs}\ Q)]\!]$
$= \lambda i^* .\ \textbf{if}\ (length\ i^*) = \mathcal{N}[\![N_{numargs}]\!]$
$\textbf{then}\ (resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ (Value^* \mapsto Stack\ (map\ Int \mapsto Value\ i^*)))))$
$\textbf{else}\ errorAnswer\ \textbf{fi}$

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

$\mathcal{C}[\![N]\!] = (push\ (Value \mapsto Result\ (Int \mapsto Value\ (\mathcal{N}[\![N]\!]))))$
$\mathcal{C}[\![(\,Q)]\!] = (push\ (Value \mapsto Result\ (StackTransform \mapsto Value\ \mathcal{Q}[\![Q]\!])))$
$\mathcal{C}[\![\texttt{pop}]\!] = pop$
$\mathcal{C}[\![\texttt{swap}]\!] = \lambda s .\ (push\ (top\ (pop\ s))\ (push\ (top\ s)\ (pop\ (pop\ s))))$
$\mathcal{C}[\![\texttt{nget}]\!] = \lambda s .\ \textbf{matching}\ (top\ s)$
$\triangleright\ (Value \mapsto Result\ (Int \mapsto Value\ i))\ [\![\ (push\ (intAt\ i\ (pop\ s))\ (pop\ s))$
$\triangleright\ \textbf{else}\ errorStack\quad \textbf{endmatching}$
$\mathcal{C}[\![\texttt{sel}]\!] = \lambda s .\ \textbf{matching}\ (top\ (pop\ (pop\ s)))$
$\triangleright\ (Value \mapsto Result\ (Int \mapsto Value\ i))\ [\![$
$(push\ (\textbf{if}\ (i =_{Int} 0)\ \textbf{then}\ (top\ s)\ \textbf{else}\ (top\ (pop\ s))\ \textbf{fi})$
$(pop\ (pop\ (pop\ s)))\ )$
$\triangleright\ \textbf{else}\ errorStack\quad \textbf{endmatching}$
$\mathcal{C}[\![\texttt{exec}]\!] = \lambda s .\ (transform\ (top\ s)\ (pop\ s))$
$\mathcal{C}[\![A]\!] = (arithop\ \mathcal{A}[\![A]\!])$
$\mathcal{C}[\![R]\!] = (arithop\ (\lambda i_1\, i_2 .\ (Value \mapsto Result$
$(Int \mapsto Value\ (\textbf{if}\ (\mathcal{R}[\![R]\!]\ i_1\ i_2)\ \textbf{then}\ 1\ \textbf{else}\ 0\ \textbf{fi})))))$

$\mathcal{A}[\![\texttt{sub}]\!] = \lambda i_1\, i_2 .\ (Value \mapsto Result\ (Int \mapsto Value\ (i_1 -_{Int} i_2)))$
Similarly for `add`, `mul`
$\mathcal{A}[\![\texttt{div}]\!] = \lambda i_1\, i_2 .\ \textbf{if}\ (i_2 =_{Int} 0)\ \textbf{then}\ errorResult$
$\textbf{else}\ (Value \mapsto Result\ (Int \mapsto Value\ (i_1 /_{Int} i_2)))\ \textbf{fi}$

$\mathcal{R}[\![\texttt{lt}]\!] = <_{Int}$
Similarly for `eq` and `gt`

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

Figure 4.13: Valuation functions for POSTFIX.

resulting stack. The definitions of *resToAns* and *top* guarantee that an error answer is returned when the stack is empty or does not have an integer as its top element. An error is also signaled when the number of arguments does not match the expected number $N_{numargs}$.

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 function $\mathcal{C}$ are straightforward. The integers and transforms corresponding to numerals and executable sequences are simply pushed onto the stack after appropriate injections into the *Value* and *Result* domains.[3] The transform associated with the `pop` command is simply the *pop* auxiliary function, while the transform associated with `swap` is expressed as a composition of *push*, *top*, and *pop*. If the top stack element is an integer i, the `nget` transform replaces it by the ith element from the rest of the stack if that element is an integer; in all other cases, `nget` returns an error stack. The `sel` transform selects one of the top two stack elements based on the numeric value of the third stack element; an error is signaled 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 signaled. 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 the PostFix 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

[3] 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.

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, unraveling such a 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 $Int \rightarrow Int \rightarrow Result$, 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,

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

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 $Result \rightarrow StackTransform$. Since

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

is clearly an element of *Result*, 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:

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

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.

Figure 4.14 illustrates using the PostFix denotational semantics to determine the result of applying the program `(postfix 2 3 sub swap pop)` to the argument integers $[7, 8]$. To make the figure more concise, we use the shorthand

Note: $\hat{n}$ is a shorthand for $(Int \mapsto Value\ n)$

$\mathcal{P}[\![$`(postfix 2 3 sub swap pop)`$]\!]\ [7,8]$

$= \textbf{if}\ (length\ [7,8]) = \mathcal{N}[\![2]\!]$
$\quad \textbf{then}\ resToAns\ \big(top\ \big(\mathcal{Q}[\![$`3 sub swap pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{7},\hat{8}])\big)\big)$
$\quad \textbf{else}\ errorAnswer\ \textbf{fi}$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`3 sub swap pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{7},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big((\mathcal{Q}[\![$`sub swap pop`$]\!]) \circ (\mathcal{C}[\![$`3`$]\!])\big)\ (Value^* \mapsto Stack\ [\hat{7},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`sub swap pop`$]\!]\ \big(\mathcal{C}[\![$`3`$]\!]\ (Value^* \mapsto Stack\ [\hat{7},\hat{8}])\big)\big)\big)$

$= resToAns\ (top\ ((\mathcal{Q}[\![$`sub swap pop`$]\!])\ (push\ (Value \mapsto Result\ \hat{3})$
$\qquad (Value^* \mapsto Stack\ [\hat{7},\hat{8}]))))$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`sub swap pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{3},\hat{7},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big((\mathcal{Q}[\![$`swap pop`$]\!]) \circ (\mathcal{C}[\![$`sub`$]\!])\big)\ (Value^* \mapsto Stack\ [\hat{3},\hat{7},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`swap pop`$]\!]\ \big(\mathcal{C}[\![$`sub`$]\!]\ (Value^* \mapsto Stack\ [\hat{3},\hat{7},\hat{8}])\big)\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`swap pop`$]\!]\ \big(arithop\ (\mathcal{A}[\![$`sub`$]\!])\ (Value^* \mapsto Stack\ [\hat{3},\hat{7},\hat{8}])\big)\big)\big)$

$= resToAns\ (top\ (\mathcal{Q}[\![$`swap pop`$]\!]\ (push\ (Value \mapsto Result\ (7\ \widehat{-_{Int}}\ 3))$
$\qquad (Value^* \mapsto Stack\ [\hat{8}]))))$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`swap pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big((\mathcal{Q}[\![$`pop`$]\!]) \circ (\mathcal{C}[\![$`swap`$]\!])\big)\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`pop`$]\!]\ \big(\mathcal{C}[\![$`swap`$]\!]\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)\big)$

$= resToAns\ (top\ (\mathcal{Q}[\![$`pop`$]\!]\ (push\ \big(top\ \big(pop\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)$
$\qquad (push\ \big(top\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)$
$\qquad\quad \big(pop\ \big(pop\ (Value^* \mapsto Stack\ [\hat{4},\hat{8}])\big)\big)\big)\big)\big)\big)\big)$

$= resToAns\ (top\ (\mathcal{Q}[\![$`pop`$]\!]\ (push\ \big(top\ (Value^* \mapsto Stack\ [\hat{8}])\big)$
$\qquad (push\ (Value \mapsto Result\ \hat{4})$
$\qquad\quad (Value^* \mapsto Stack\ [\,])\big)\big)\big)\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`pop`$]\!]\ \big(push\ (Value \mapsto Result\ \hat{8})\ (Value^* \mapsto Stack\ [\hat{4}])\big)\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![$`pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{8},\hat{4}])\big)\big)$

$= resToAns\ \big(top\ \big((\mathcal{Q}[\![]\!]) \circ (\mathcal{C}[\![$`pop`$]\!])\big)\ (Value^* \mapsto Stack\ [\hat{8},\hat{4}])\big)\big)$

$= resToAns\ \big(top\ \big(\mathcal{Q}[\![]\!]\ \big(\mathcal{C}[\![$`pop`$]\!]\ (Value^* \mapsto Stack\ [\hat{8},\hat{4}])\big)\big)\big)$

$= resToAns\ \big(top\ \big((\lambda s\,.\,s)\ (Value^* \mapsto Stack\ [\hat{4}])\big)\big)$

$= resToAns\ \big(top\ (Value^* \mapsto Stack\ [\hat{4}])\big)$

$= resToAns\ (Value \mapsto Result\ \hat{4})$

$= (Int \mapsto Answer\ 4)$

Figure 4.14: Equational proof that applying the POSTFIX program `(postfix 2 3 sub swap pop)` to the arguments $[7,8]$ yields the answer 4.

$\hat{n}$ to stand for $(\mathit{Int} \mapsto \mathit{Value}\ n)$. Each line of the equational proof is justified by simple mathematical reasoning. For example, the equality

$$\begin{aligned}&\mathit{resToAns}\ \big(\mathit{top}\ \big(((\mathcal{Q}[\![\texttt{pop}]\!])\circ(\mathcal{C}[\![\texttt{swap}]\!]))\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4},\hat{8}])\big)\big)\\ =\ &\mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{pop}]\!]\ \big(\mathcal{C}[\![\texttt{swap}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4},\hat{8}])\big)\big)\big)\end{aligned}$$

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

$$\begin{aligned}&\mathit{resToAns}\ (\mathit{top}\ (\mathcal{Q}[\![\texttt{swap pop}]\!]\ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ \widehat{(7 -_{\mathit{Int}} 3)})\\ &\qquad\qquad\qquad\qquad (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{8}])))))\\ =\ &\mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4},\hat{8}])\big)\big)\end{aligned}$$

is justified by the definition of $-_{\mathit{Int}}$ and the specification for the *push* function. The proof shows that the result of the program execution is the integer 4.

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 of a programming language procedure is a theorem. For example, it's not difficult to prove a theorem stating that for any numeral N, any command sequence Q, and any stack s, the following equality is valid:

$$\begin{aligned}&(\mathcal{Q}[\![N\ .\ Q]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ v^*))\\ =&(\mathcal{Q}[\![Q]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ ((\mathit{Int} \mapsto \mathit{Value}\ \mathcal{N}[\![N]\!])\ .\ v^*))).\end{aligned}$$

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

$$\begin{aligned}&\big(\mathcal{Q}[\![\texttt{3 sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{7},\hat{8}])\big)\\ =\ &\big(\mathcal{Q}[\![\texttt{sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{3},\hat{7},\hat{8}])\big)\end{aligned}$$

A few such theorems can greatly reduce the length of the sample proof. In fact, if we prove other theorems analogous to the operational rules, we can obtain a proof whose structure closely corresponds to the configuration sequence for an operational execution of the program (see Figure 4.15).

Figure 4.16 shows how the equational proof in Figure 4.15 can be generalized two handle two arbitrary integer arguments. Based on this result, we conclude that the meaning of the POSTFIX program `(postfix 2 3 sub swap pop)` is:

$$\begin{aligned}&\mathcal{P}[\![\texttt{(postfix 2 3 sub swap pop)}]\!]\\ =\ &\lambda i^*\ .\ \textbf{matching}\ i^*\\ &\qquad \triangleright\ [i_1, i_2]\ [\!]\ (\mathit{Int} \mapsto \mathit{Answer}\ (i_1 -_{\mathit{Int}} 3))\\ &\qquad \triangleright\ \textbf{else}\ \mathit{errorAnswer}\ \textbf{endmatching}\ .\end{aligned}$$

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub swap pop)}]\!]\ [7,8] \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{3 sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{7},\hat{8}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{3},\hat{7},\hat{8}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4},\hat{8}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![\texttt{pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{8},\hat{4}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ \big(\mathcal{Q}[\![]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4}])\big)\big) \\
= \mathit{resToAns}\ \big(\mathit{top}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{4}])\big) \\
= \mathit{resToAns}\ (\mathit{Value} \mapsto \mathit{Result}\ \hat{4}) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ 4)
\end{array}
$$

Figure 4.15: Alternative equational proof with an operational flavor.

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub swap pop)}]\!]\ [i_1, i_2] \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{3 sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{i_1}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{sub swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{3}, \hat{i_1}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{swap pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{pop}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{i_2}, \widehat{(i_1\ -_{\mathit{Int}}\ 3)}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}])\Big) \\
= \mathit{resToAns}\ (\mathit{Value} \mapsto \mathit{Result}\ \widehat{(i_1\ -_{\mathit{Int}}\ 3)}) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ (i_1\ -_{\mathit{Int}}\ 3))
\end{array}
$$

Figure 4.16: Version of equational proof for two arbitrary integer arguments.

▷ **Exercise 4.4** Use the PostFix denotational semantics to determine the values of the PostFix programs in Exercise 1.1. ◁

▷ **Exercise 4.5** Modify the PostFix denotational semantics to handle PostFix2. Include valuation functions for `:`, `(skip)`, and `(exec)`. ◁

▷ **Exercise 4.6** For each of the following, modify the PostFix denotational semantics to handle the specified extensions:

a. The `pair`, `left`, and `right` commands from Exercise 3.36.

b. The `for` and `repeat` commands from Exercise 3.39.

c. The *I*, `def`, and `get` commands from Exercise 3.43. ◁

4.3.3 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 4.11. Figure 4.17 presents one implementation of the specifications. As an exercise, you should make sure that these definitions type check, and that they satisfy the specifications in Figure 4.11.

Notice that several functions in Figure 4.17 describe similar manipulations. *push*, *pop*, and *arithop* all check to see if their input stack is an error stack. If so, they return *errorStack*; if not, they perform some manipulation on the sequence of values in the stack. We can abstract over these similarities by introducing three abstractions (Figure 4.18) similar to the *with-int* error hiding function defined in the EL denotational semantics:

- *with-stack-values* takes a function f from value sequences to stacks and returns a stack transform that (1) maps a non-error stack to the result of applying f to the value sequence in the stack, and (2) maps an error stack to an error stack.

- *with-val&stack* takes a function f from a value to a stack transform and returns a stack transform that (1) maps any stack whose value sequence consists of the value v followed by v_{rest}* to the result of applying f to v and the stack whose values are v_{rest}*, and (2) maps any stack not of this form to the error stack.

- *with-int&stack* takes a function f from an integer to a stack transform and returns a stack transform that (1) maps any stack whose value sequence consists of an integer i followed by v_{rest}* to the result of applying f to v

$$
\begin{array}{l}
\mathit{empty\text{-}stack} : \mathit{Stack} \ = (\mathit{Value}^* \mapsto \mathit{Stack}\ [\,]_{\mathit{Value}}) \\
\mathit{errorStack} : \mathit{Stack} \ = (\mathit{Error} \mapsto \mathit{Stack}\ \mathit{error}) \\
\mathit{errorTransform} : \mathit{StackTransform} \ = \lambda s\,.\,\mathit{errorStack} \\
\mathit{errorResult} : \mathit{Result} \ = (\mathit{Error} \mapsto \mathit{Result}\ \mathit{error}) \\
\mathit{errorAnswer} : \mathit{Answer} \ = (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error}) \\
\\
\mathit{push} : \mathit{Result} \to \mathit{StackTransform} \\
= \lambda rs\,.\ \textbf{matching}\ \langle r, s\rangle \\
\qquad \triangleright \langle (\mathit{Value} \mapsto \mathit{Result}\ v), (\mathit{Value}^* \mapsto \mathit{Stack}\ v^*)\rangle \ [\!]\ (v\ .\ v^*) \\
\qquad \triangleright (\mathit{Error} \mapsto \mathit{Result}\ \mathit{error}) \ [\!]\ \mathit{errorStack}\ \ \textbf{endmatching} \\
\\
\mathit{pop} : \mathit{StackTransform} \\
= \lambda s\,.\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ (v_{\mathit{head}}\ .\ v_{\mathit{tail}}{}^*)) \ [\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ v_{\mathit{tail}}{}^*) \\
\qquad \triangleright \textbf{else}\ \mathit{errorStack}\ \ \textbf{endmatching} \\
\\
\mathit{top} : \mathit{Stack} \to \mathit{Result} \\
= \lambda s\,.\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ (v_{\mathit{head}}\ .\ v_{\mathit{tail}}{}^*)) \ [\!]\ (\mathit{Value} \mapsto \mathit{Result}\ v_{\mathit{head}}) \\
\qquad \triangleright \textbf{else}\ \mathit{errorResult}\ \ \textbf{endmatching} \\
\\
\mathit{intAt} : \mathit{Int} \to \mathit{Stack} \to \mathit{Result} \\
= \lambda is\,.\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ v^*) \ [\!] \\
\qquad\ \ \textbf{if}\ (1 \leq_{\mathit{Int}} i_{\mathit{index}})\ \ \textbf{and}\ (i_{\mathit{index}} \leq_{\mathit{Int}} (\mathit{length}\ v^*)) \\
\qquad\ \ \textbf{then matching}\ (\mathit{nth}\ i\ v^*) \\
\qquad\qquad \triangleright (\mathit{Int} \mapsto \mathit{Value}\ i_{\mathit{result}}) \ [\!]\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ i_{\mathit{result}})) \\
\qquad\qquad \triangleright \textbf{else}\ \mathit{errorResult} \\
\qquad\ \ \textbf{else}\ \mathit{errorResult}\ \textbf{fi} \\
\qquad \triangleright \textbf{else}\ \mathit{errorResult}\ \ \textbf{endmatching} \\
\\
\mathit{arithop} : (\mathit{Int} \to \mathit{Int} \to \mathit{Result}) \to \mathit{StackTransform} \\
= \lambda f\, s\,.\ \textbf{matching}\ s \\
\qquad \triangleright (\mathit{Value}^* \mapsto \mathit{Stack}\ ((\mathit{Int} \mapsto \mathit{Value}\ i_1)\ .\ (\mathit{Int} \mapsto \mathit{Value}\ i_2)\ .\ v_{\mathit{rest}}{}^*)) \ [\!] \\
\qquad\ \ (\mathit{push}\ \ (f\ \ i_2\ \ i_1)\ \ v_{\mathit{rest}}{}^*) \\
\qquad \triangleright \textbf{else}\ \mathit{errorStack}\ \ \textbf{endmatching} \\
\\
\mathit{transform} : \mathit{Result} \to \mathit{StackTransform} \\
= \lambda r\,.\ \textbf{matching}\ r \\
\qquad \triangleright (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{StackTransform} \mapsto \mathit{Value}\ t)) \ [\!]\ t \\
\qquad \triangleright \textbf{else}\ \mathit{errorTransform}\ \ \textbf{endmatching} \\
\\
\mathit{resToAns} : \mathit{Result} \to \mathit{Answer} \\
= \lambda r\,.\ \textbf{matching}\ r \\
\qquad \triangleright (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ i)) \ [\!]\ (\mathit{Int} \mapsto \mathit{Answer}\ i) \\
\qquad \triangleright \textbf{else}\ \mathit{errorAnswer}\ \ \textbf{endmatching}
\end{array}
$$

Figure 4.17: Functions manipulating the semantic domains for PostFix.

```
with-stack-values : (Value* → Stack) → StackTransform
= λf s . matching s
        ▷ (Value* ↦ Stack v*) ‖ (f v*)
        ▷ else errorStack endmatching

with-val&stack : (Value → StackTransform) → StackTransform
= λf . (with-stack-values
          (λv* . matching v*
                  ▷ v1 . v_rest* ‖ (f v1 (Value* ↦ Stack v_rest*))
                  ▷ else errorStack endmatching ))

with-int&stack : (Int → StackTransform) → StackTransform
= λf . (with-val&stack
          (λv . matching v
                  ▷ (Int ↦ Value i) ‖ (f i)
                  ▷ else errorTransform endmatching ))

push : Result → StackTransform
= λr . matching r
       ▷ (Value ↦ Result v) ‖ (with-stack-values (λv* . (Value* ↦ Stack (v . v*))))
       ▷ else errorTransform
       endmatching

pop : StackTransform = with-val&stack (λv_hd s_tl . s_tl)

arithop : (Int → Int → Result) → StackTransform
= λf . (with-int&stack (λi1 . (with-int&stack (λi2 . (push (f i2 i1)))))))
```

Figure 4.18: The auxiliary functions *with-stack-values*, *with-val&stack*, and *with-integer&stack* simplify some of the semantic functions for POSTFIX. (Only the modified functions are shown.)

and the stack whose values are v_{rest}*, and (2) 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 4.18, rewriting *push* in terms of *with-stack-values* removes an error check from the definition. Using *with-val&stack* and *with-int&stack* greatly simplify *pop* and *arithop*; the updated versions concisely capture the essence of these functions 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-int&stack* is a stack transform t, and t is equivalent to $\lambda s\,.\,(t\ s)$, the new version of *arithop* can be rewritten as:

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

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

Even more important is recognizing the pattern

$$((\textit{with-int\&stack}\ \ (\lambda i s_{rest}\ .\ E))\ \ s)$$

as a construct that binds names to values. This pattern can be pronounced as:

> "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."

Some of the PostFix valuation functions can be re-expressed using the error hiding functions directly. For example, the valuation clause for `swap` can be written as:

$$\mathcal{C}[\![\texttt{swap}]\!] = \textit{with-val\&stack}\ \ (\lambda v_1\ .\ (\textit{with-val\&stack}\ \ (\lambda v_2\ .(\textit{push}\ \ v_2)\ \circ\ (\textit{push}\ \ v_1))))$$

You should convince yourself that this has the same meaning as the version written using *push*, *top*, and *pop*.

▷ **Exercise 4.7**

a. By analogy with *with-int&stack*, define a function *with-trans&stack* whose signature is $(StackTransform \to StackTransform) \to StackTransform$.

b. Rewrite the valuation clauses for the commands `nget`, `sel`, and `exec` using *with-val&stack*, *with-int&stack*, and *with-trans&stack* to eliminate all occurrences of *top*, *pop*, *transform*, and **matching**. ◁

4.4 Denotational Reasoning

The denotational definitions of EL and PostFix presented in the previous section are mathematically elegant, but how useful are they? We have already shown how they can be used to determine the meanings of particular programs. In this section we show how denotational semantics helps us to reason about program equality and safe program transformations. The compositional structure of the denotational semantics makes it more amenable to proving certain properties than the operational semantics. We also study the relationship between operational semantics and denotational semantics.

4.4.1 Program Equality

Above, we studied the PostFix program `(postfix 2 3 sub swap pop)`, which takes two integer arguments and returns three less than the first argument:

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub swap pop)}]\!] \\
= \lambda i^* .\ \textbf{matching}\ i^* \\
\qquad \triangleright\ [i_1, i_2]\ [\![\ \ (Int \mapsto Answer\ (i_1 -_{Int} 3)) \\
\qquad \triangleright\ \textbf{else}\ \mathit{errorAnswer}\ \textbf{endmatching}\,.
\end{array}
$$

Intuitively, the purpose of the `swap pop` is to get rid of the second argument, which is ignored by the program. But in a PostFix program, only the integer at the top of the final stack can be observed and any other stack values are ignored. So we should be able to remove the `swap pop` from the program without changing its behavior.

We can formalize this reasoning using denotational semantics. Figure 4.19 shows a derivation of the meaning of the program `(postfix 2 3 sub)` when it is applied to two arguments. From this, we deduce that the meaning of `(postfix 2 3 sub)` is:

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub)}]\!] \\
= \lambda i^* .\ \textbf{matching}\ i^* \\
\qquad \triangleright\ [i_1, i_2]\ [\![\ \ (Int \mapsto Answer\ (i_1 -_{Int} 3)) \\
\qquad \triangleright\ \textbf{else}\ \mathit{errorAnswer}\ \textbf{endmatching}\,.
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(postfix 2 3 sub)}]\!]\ [i_1, i_2] \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{3 sub}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{i_1}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![\texttt{sub}]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\hat{3}, \hat{i_1}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ \Big(\mathcal{Q}[\![]\!]\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}, \hat{i_2}])\Big)\Big) \\
= \mathit{resToAns}\ \Big(\mathit{top}\ (\mathit{Value}^* \mapsto \mathit{Stack}\ [\widehat{(i_1\ -_{\mathit{Int}}\ 3)}, \hat{i_2}])\Big) \\
= \mathit{resToAns}\ (\mathit{Value} \mapsto \mathit{Result}\ \widehat{(i_1\ -_{\mathit{Int}}\ 3)}) \\
= (\mathit{Int} \mapsto \mathit{Answer}\ (i_1\ -_{\mathit{Int}}\ 3))
\end{array}
$$

Figure 4.19: The meaning of `(postfix 2 3 sub)` on two arguments.

Since `(postfix 2 3 sub)` and `(postfix 2 3 sub swap pop)` have exactly the same meaning, they cannot be distinguished as programs.

Denotational semantics can also be used to show that programs from different languages have the same meaning. For example, it is not hard to show that the meaning of the EL program `(el 2 (- (arg 1) 3))` is:

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{(el 2 (- (arg 1) 3)}]\!] \\
= \lambda i^*.\ \textbf{matching}\ i^* \\
\qquad \triangleright\ [i_1, i_2]\ ]\!]\ \ (\mathit{Int} \mapsto \mathit{Answer}\ (i_1\ -_{\mathit{Int}}\ 3)) \\
\qquad \triangleright\ \textbf{else}\ \mathit{errorAnswer}\ \textbf{endmatching}\,.
\end{array}
$$

If you review the semantic domains for EL and PostFix, you will see that the *Answer* domain is the same for both languages. So the above fact means that this EL program is interchangeable with the two PostFix programs whose meanings are given above.

4.4.2 Safe Transformations: A Denotational Approach

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 $Q_1 =_{obs} Q_2$.

This theorem is a consequence of a so-called adequacy property of PostFix, which we will study later in Section 4.4.4.2.

We can use this theorem to help us prove the behavioral equivalence of two command sequences. For instance, consider the pair of command sequences $[\texttt{1}, \texttt{add}, \texttt{2}, \texttt{add}]$ and $[\texttt{3}, \texttt{add}]$. Figure 4.20 shows that these are denotationally equivalent, so, by the above theorem, they must be observationally equivalent. The equational reasoning in Figure 4.20 uses the following three equalities, whose proofs are left as exercises:

$$(\mathcal{Q}[\![C_1 \;\; C_2 \;\; \ldots C_n]\!]) = (\mathcal{C}[\![C_n]\!]) \circ \ldots \circ (\mathcal{C}[\![C_2]\!]) \circ (\mathcal{C}[\![C_1]\!]) \tag{4.1}$$

$$(\mathit{with\text{-}int\&stack}\ f) \circ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ i))) = (f\ \ i) \tag{4.2}$$

$$t \circ (\mathit{with\text{-}int\&stack}\ f) = (\mathit{with\text{-}int\&stack}\ (\lambda i\,.\,(t \circ (f\ \ i)))) \tag{4.3}$$

where t maps *errorStack* to *errorStack*

It is worth noting that the denotational proof that $[\texttt{1}, \texttt{add}, \texttt{2}, \texttt{add}] =_{obs} [\texttt{3}, \texttt{add}]$ has a very different flavor than the operational proof of this fact given in Section 3.4.4. The operational proof worked by case analysis on the initial stack. The denotational proof in Figure 4.20 works purely by equational reasoning — there is no hint of case analysis here. This is because the all the case analyses are hidden within the carefully chosen abstractions *with-int&stack* and *push* and equalities (4.1)–(4.3). The case analyses would become apparent if these were expanded to show explicit **matching** expressions.

Denotational justifications for the safety of transformations are not limited to PostFix. For example, Figure 4.21 shows that EL numerical expressions `(+ NE NE)` and `(* 2 NE)` have the same meaning. So one can safely be substituted for the other in any EL program without changing the meaning of the program.

▷ **Exercise 4.8**

a. Prove equalities (4.1)–(4.3).

b. Equality (4.3) requires that t maps *errorStack* to *errorStack*. Show that the equality is not true if this requirement is violated. ◁

$$
\begin{array}{l}
(\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}]\!]) \text{ , by (4.1)} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_1\,' \,.\, (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2\,' \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2\,' + i_1\,')))))))) \\
\quad \circ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (\mathcal{N}[\![\texttt{2}]\!])))) \\
\quad \circ (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_1 \,.\, (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + i_1)))))))) \\
\quad \circ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (\mathcal{N}[\![\texttt{1}]\!])))) \text{ , by definition of } \mathcal{C} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_2\,' \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2\,' + 2))))) \\
\quad \circ \ (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + 1)))))) \text{ , by (4.2)} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_2 \,.\, \ (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2\,' \,.\, \ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2\,' + 2)))))) \\
\qquad \circ \ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + 1)))))), \text{ by (4.3)} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ ((i_2 + 1)\ + 2))))) \text{ , by (4.2)} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + 3)))))) \text{ , by definition of } +_{\mathit{Int}} \\
= (\mathit{with\text{-}int\&stack} \\
\quad (\lambda i_3 \,.\, (\mathit{with\text{-}int\&stack} \\
\qquad (\lambda i_2 \,.\, (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (i_2 + i_3))))))) \\
\quad \circ (\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ \ (\mathcal{N}[\![\texttt{3}]\!])))) \text{ , by (4.2)} \\
= (\mathcal{C}[\![\texttt{add}]\!]) \circ (\mathcal{C}[\![\texttt{3}]\!]) \text{ , by definition of } \mathcal{C} \\
= (\mathcal{Q}[\![\texttt{3 add}]\!]) \text{ , by (4.1)}
\end{array}
$$

Figure 4.20: 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.

$$
\begin{aligned}
&\mathcal{NE}[\![\texttt{(+ } NE\ \ NE\texttt{)}]\!] \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_1\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_2\,.\,(\mathcal{A}[\![\texttt{+}]\!]\ \ i_1\ \ i_2))) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (i_2 +_{Int} i_2) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (2 \times_{Int} i_2) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_2\,.\,(\mathcal{A}[\![\texttt{+}]\!]\ \ i_2\ \ i_2)) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_2\,.\,(\mathcal{A}[\![\texttt{*}]\!]\ \ 2\ \ i_2)) \\
&= \lambda i^*\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![\texttt{2}]\!]\ \ i^*)\ \ (\lambda i_1\,.\,\mathit{with\text{-}int}\ \ (\mathcal{NE}[\![NE]\!]\ \ i^*)\ \ (\lambda i_2\,.\,(\mathcal{A}[\![\texttt{*}]\!]\ \ i_1\ \ i_2))) \\
&= \mathcal{NE}[\![\texttt{(* 2 } NE\texttt{)}]\!]
\end{aligned}
$$

Figure 4.21: Denotational proof that `(+ NE NE)` may safely be replaced by `(* 2 NE)` in EL.

▷ **Exercise 4.9**

a. We have seen that `(postfix 2 3 sub swap pop)` and `(postfix 2 3 sub)` are equivalent programs. But in general it is *not* safe to replace the command sequence `3 sub swap pop` by `3 sub`. Give a context in which this replacement would change the meaning of a program.

b. Use denotational reasoning to show that it is safe to replace any of the following command sequences by `3 sub swap pop`:

   i. `swap pop 3 sub`

   ii. `(3 sub) swap pop exec`

   iii. `3 2 nget swap sub swap pop swap pop` ◁

▷ **Exercise 4.10** Use the PostFix denotational semantics to either prove or disprove the purported observational equivalences in Exercise 3.28. ◁

▷ **Exercise 4.11** Use the EL denotational semantics to either prove or disprove the safety of the EL transformations in Exercise 3.32. ◁

4.4.3 Technical Difficulties

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:

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

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 *StackTransform*, *Stack*, and *Value* domains are mutually recursive:

$$\begin{aligned} \mathit{StackTransform} &= \mathit{Stack} \to \mathit{Stack} \\ \mathit{Stack} &= \mathit{Value}^* + \mathit{Error} \\ \mathit{Value} &= \mathit{Int} + \mathit{StackTransform} \end{aligned}$$

It turns out that solving such recursive domain equations sometimes requires extending some domains with an element that models nontermination, written $\perp$ and pronounced "bottom." We will study this element in more detail in the next chapter, where it plays a prominent role. In the case of PostFix, it turns out that both the *Stack* and *Answer* domains must include $\perp$, and this is able to model the meaning of non-terminating command sequences.

4.4.4 Relating Operational and Denotational Semantics

We have presented the operational and denotational semantics of several simple languages, but have not studied the connection between them. What is the relationship between these two forms of semantics? How can we be sure that reasoning done with one form of semantics is valid in the other?

4.4.4.1 Soundness

Assume that an operational semantics has a deterministic behavior function of the form

$$\mathit{beh}_{\mathit{det}} : (\text{Program} \times \text{Inputs}) \to \text{Outcome}$$

and that the related denotational semantics has a meaning function

$$\mathit{meaning} : (\text{Program} \times \mathit{Args}) \to \mathit{Answer},$$

where *Args* is a domain of program arguments and *Answer* is the domain of final answers. Also suppose that there is a function *in* that maps between the syntactic and semantic input domains and a function *out* that maps between the syntactic and semantic output domains:

$$\begin{aligned} \mathit{in} &: \text{Inputs} \to \mathit{Args} \\ \mathit{out} &: \text{Outcome} \to \mathit{Answer}. \end{aligned}$$

Then we define the following notion of soundness:

$I \in \text{Inputs} = \text{Intlit}^*$
$o \in \text{Outcome} = \text{Intlit} + \text{StuckOut}$
$\text{StuckOut} = \{\texttt{stuckout}\}$
$ar \in \mathit{Args} = \mathit{Int}^*$
$a \in \mathit{Answer} = \mathit{Int} + \mathit{Error}$
$\mathit{Error} = \{\mathit{error}\}$

$\mathit{in} : \text{Inputs} \to \mathit{Args}$
$\mathit{in} = \lambda N^* .\ (\mathit{map}\ \mathcal{N}\ N^*)$

$\mathit{out} : \text{Outcome} \to \mathit{Answer}$
$\mathit{out} = \lambda o .\ \textbf{matching}\ o$
$\triangleright\, (\text{Intlit} \mapsto \text{Outcome}\ N)\ [\!]\ (\mathit{Int} \mapsto \mathit{Answer}\ (\mathcal{N}[\![N]\!]))$
$\triangleright\, \textbf{else}\ (\mathit{Error} \mapsto \mathit{Answer}\ \mathit{error})\ \textbf{endmatching}$

$\mathit{beh}_{\mathit{det}} : (\text{Program} \times \text{Inputs}) \to \text{Outcome}$
$\mathit{beh}_{\mathit{det\,EL}}$ is defined in Exercise 3.10
and $\mathit{beh}_{\mathit{det\,PostFix}}$ is defined in Section 3.2.2.

$\mathit{meaning} : (\text{Program} \times \mathit{Args}) \to \mathit{Answer}$
$\mathit{meaning}_{EL} = \lambda\langle P, ar\rangle .\ (\mathcal{P}_{EL}[\![P]\!]\ ar)$, where $\mathcal{P}_{EL}$ is defined in Section 4.2.4.
$\mathit{meaning}_{PostFix} = \lambda\langle P, ar\rangle .\ (\mathcal{P}_{PostFix}[\![P]\!]\ ar)$,
where $\mathcal{P}_{PostFix}$ is defined in Section 4.3.2.

Figure 4.22: Instantiation of soundness components for EL and PostFix.

> **Denotational Soundness:** A denotational semantics is **sound with respect to (wrt)** an operational semantics if for all programs P and inputs I,
>
> $$\mathit{meaning}\ \langle P, (\mathit{in}\ I)\rangle = (\mathit{out}\ (\mathit{beh}_{\mathit{det}}\ \langle P, I\rangle)) .$$

This definition says that the denotational semantics agrees with the operational semantics on the result of executing a program on any given inputs. Figure 4.22 shows how the parts of the soundness definition can be instantiated for EL and PostFix.

We will now sketch a proof that the denotational semantics for PostFix is sound wrt the operational semantics for PostFix. The details of this proof, and a denotational soundness proof for EL, are left as exercises. The essence of the denotational soundness proof for PostFix is to define the meaning of an operational configuration, and show that each step in the PostFix SOS preserves this meaning. Recall that a configuration in the PostFix SOS has the form Commands × Stack, where

$\mathcal{V}$: Value $\rightarrow$ *Value*
‘ $= \lambda V$. **matching** V
$\triangleright$ (Intlit $\mapsto$ Value N) $[\!]$ $(Int \mapsto Value\ (\mathcal{N}[\![N]\!]))$
$\triangleright$ (Commands $\mapsto$ Value Q) $[\!]$ $(StackTransform \mapsto Value\ (\mathcal{Q}[\![Q]\!]))$
endmatching

$\mathcal{S}$: Stack $\rightarrow$ *Stack* $= \lambda V^*$. $(Value^* \mapsto Stack\ (map\ \mathcal{V}\ V^*))$

$\mathcal{CF}$: Commands $\times$ Stack $\rightarrow$ *Answer* $= \lambda\langle Q, S\rangle$. $resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ (\mathcal{S}[\![S]\!])))$

Figure 4.23: Meaning of a PostFix configuration.

$S \in$ Stack $=$ Value*
$V \in$ Value $=$ Intlit $+$ Commands

Figure 4.23 defines a function $\mathcal{CF}$ that maps an operational configuration to an element of *Answer*. We establish the following lemmas:

1. For any PostFix program $P =$ `(postfix` $N_{numargs}$ Q`)` and numerals N^*,

$$(\mathcal{P}[\![P]\!]\ (in\ N^*)) = \mathcal{CF}[\![(IF\ \langle P, N^*\rangle)]\!],$$

where *IF* is the input function defined in Figure 3.3 that maps a PostFix program and inputs into an initial SOS configuration. There are two cases:

(a) When $\mathcal{N}[\![N_{numargs}]\!] = (length\ N^*)$, both the left and right hand sides of the equation denote

$$resToAns\,(top\ (\mathcal{Q}[\![Q]\!]\ (Value^* \mapsto Stack\ (map\ (Int \mapsto Value \circ \mathcal{N})\ N^*)))).$$

(b) When $\mathcal{N}[\![N_{numargs}]\!] \neq (length\ N^*)$, the left hand side of the equation denotes *errorAnswer* and the right hand side denotes

$$\begin{aligned}
&\mathcal{CF}[\![(IF\ \langle P, N^*\rangle)]\!]\\
&= \mathcal{CF}[\![\langle [\,]_{\text{Commands}}, [\,]_{\text{Stack}}\rangle]\!]\\
&= resToAns\ (top\ (\mathcal{Q}[\![[\,]_{\text{Commands}}]\!]\ (Value^* \mapsto Stack\ [\,]_{Stack})))\\
&= resToAns\ (top\ (Value^* \mapsto Stack\ [\,]_{Stack}))\\
&= errorAnswer.
\end{aligned}$$

2. For any transition $cf \Rightarrow cf'$, $\mathcal{CF}[\![cf]\!] = \mathcal{CF}[\![cf']\!]$. This can be shown by demonstrating this equality for each of the PostFix transition rules in Figure 3.4. For example, one such rule is:

$$\langle \texttt{exec}\ .\ Q_{rest},\ (\ Q_{exec}\ )\ .\ S\rangle \Rightarrow \langle Q_{exec}\ @\ Q_{rest},\ S\rangle \qquad [execute]$$

For this rule we have

$$\begin{aligned}
&\mathcal{CF}[\![\langle \texttt{exec}\ .\ Q_{rest},\ (\ Q_{exec}\ )\ .\ S\rangle]\!]\\
&= resToAns\ (top\ (\mathcal{Q}[\![\texttt{exec}\ .\ Q_{rest}]\!]\ (Value^* \mapsto Stack\ (\mathcal{V}[\![(\ Q_{exec}\ )]\!]\ .\ v^*))))),\\
&\quad \text{where } v^* = (map\ \mathcal{V}\ S)\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![Q_{rest}]\!]\ (\mathcal{C}[\![\texttt{exec}]\!]\ (Value^* \mapsto Stack\ (\mathcal{V}[\![(\ Q_{exec}\ )]\!]\ .\ v^*)))))\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![Q_{rest}]\!]\\
&\qquad (transform\ (top\ (Value^* \mapsto Stack\ (\mathcal{V}[\![(\ Q_{exec}\ )]\!]\ .\ v^*)))\\
&\qquad\qquad (pop\ (Value^* \mapsto Stack\ (\mathcal{V}[\![(\ Q_{exec}\ )]\!]\ .\ v^*)))))))\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![Q_{rest}]\!]\\
&\qquad (transform\ (Value \mapsto Result\ (StackTransform \mapsto Value\ (\mathcal{Q}[\![Q_{exec}]\!])))\\
&\qquad\qquad (Value^* \mapsto Stack\ v^*))))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q_{rest}]\!]\ (\mathcal{Q}[\![Q_{exec}]\!]\ (Value^* \mapsto Stack\ v^*))))\\
&= resToAns\ (top\ ((\mathcal{Q}[\![Q_{rest}]\!] \circ \mathcal{Q}[\![Q_{exec}]\!])\ (Value^* \mapsto Stack\ v^*)))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q_{rest}\ @\ Q_{exec}]\!]\ (Value^* \mapsto Stack\ (map\ \mathcal{V}\ S))))\\
&= \mathcal{CF}[\![\langle Q_{exec}\ @\ Q_{rest},\ S\rangle]\!].
\end{aligned}$$

3. For any stuck configuration cf, $\mathcal{CF}[\![cf]\!] = errorAnswer$. This can be shown by enumerating the finite number of configuration patterns that stand for configurations in $Irreducible_{PFSOS}$, and showing that each denotes the error answer. For example, one such pattern is $\langle \texttt{swap}\ .\ Q,\ [\ V]\rangle$:

$$\begin{aligned}
&\mathcal{CF}[\![\langle \texttt{swap}\ .\ Q,\ [\ V]\rangle]\!]\\
&= resToAns\ (top\ (\mathcal{Q}[\![\texttt{swap}\ .\ Q]\!]\ (Value^* \mapsto Stack\ [\mathcal{V}\ V])))\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![Q]\!]\ (push\ (top\ (pop\ (Value^* \mapsto Stack\ [\mathcal{V}\ V])))\\
&\qquad\qquad (push\ (top\ (Value^* \mapsto Stack\ [\mathcal{V}\ V]))\\
&\qquad\qquad\quad (pop\ (pop\ (Value^* \mapsto Stack\ [\mathcal{V}\ V])))))))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ (push\ (top\ (Value^* \mapsto Stack\ [\,]))\\
&\qquad\qquad\qquad (push\ (Value \mapsto Result\ (\mathcal{V}[\![V]\!]))\\
&\qquad\qquad\qquad\quad (pop\ (Value^* \mapsto Stack\ [\,])))))))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ (push\ errorResult\\
&\qquad\qquad\qquad (push\ (\mathcal{V}[\![V]\!])\ \ errorStack))))\\
&= resToAns\ (top\ (\mathcal{Q}[\![Q]\!]\ errorStack))\\
&= resToAns\ (top\ errorStack)\\
&= errorAnswer.
\end{aligned}$$

We're now ready to put the lemmas together to show denotational soundness for a POSTFIX program (`postfix` $N_{numargs}$ Q_{body}) executed on inputs N_{inputs}*. There are two cases:

1. $\mathcal{N}[\![N_{numargs}]\!] = (length\ N_{inputs}^*)$ and the initial program configuration has a transition path to a final configuration:

$$\langle Q_{body},\ N_{inputs}^*\rangle \stackrel{*}{\Rightarrow} \langle [\,]_{\text{Commands}},\ N_{ans}\ .\ V_{rest}^*\rangle$$

In this case,

$$\begin{aligned}
&meaning\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), (in\ N_{inputs}^*)\rangle\\
&= \mathcal{P}[\![(\texttt{postfix}\ N_{numargs}\ Q_{body})]\!]\ (in\ N_{inputs}^*)\\
&= \mathcal{CF}[\![(IF\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N_{inputs}^*\rangle)]\!]\text{ , by lemma 1}\\
&= \mathcal{CF}[\![\langle Q_{body},\ (map\ \text{Intlit} \mapsto \text{Value}\ N_{inputs}^*)\rangle]\!]\\
&= \mathcal{CF}[\![\langle [\,]_{\text{Commands}},\ (\text{Intlit} \mapsto \text{Value}\ N_{ans})\ .\ V_{rest}^*\rangle]\!]\text{ , by lemma 2 on each } \Rightarrow\\
&= resToAns\\
&\quad (top\ (\mathcal{Q}[\![]\ (Value^* \mapsto Stack\ ((Int \mapsto Value\ (\mathcal{N}[\![N_{ans}]\!]))\ .\ (map\ \mathcal{V}\ V_{rest}^*)))))\\
&= resToAns\ (top\ (Value^* \mapsto Stack\ ((Int \mapsto Value\ (\mathcal{N}[\![N_{ans}]\!]))\ .\ (map\ \mathcal{V}\ V_{rest}^*))))\\
&= (Int \mapsto Answer\ (\mathcal{N}[\![N_{ans}]\!]))\\
&= (out\ (\text{Intlit} \mapsto \text{Outcome}\ N_{ans}))\\
&= (out\ (beh_{det\,PostFix}\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N_{inputs}^*\rangle)).
\end{aligned}$$

2. $\mathcal{N}[\![N_{numargs}]\!] \neq (length\ N_{inputs}^*)$ or the initial program configuration has a transition path to a stuck configuration. In these cases,

$$IF\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N^*\rangle \stackrel{*}{\Rightarrow} cf_{stuck},$$

where cf_{stuck} is a stuck configuration. Then we have:

$$\begin{aligned}
&meaning\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), (in\ N_{inputs}^*)\rangle\\
&= \mathcal{P}[\![(\texttt{postfix}\ N_{numargs}\ Q_{body})]\!]\ (in\ N_{inputs}^*)\\
&= \mathcal{CF}[\![(IF\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N_{inputs}^*\rangle)]\!]\text{ , by lemma 1}\\
&= \mathcal{CF}[\![cf_{stuck}]\!]\text{ , by lemma 2 on each } \Rightarrow\\
&= errorAnswer\text{, by lemma 3}\\
&= (out\ \texttt{stuck})\\
&= (out\ (beh_{det\,PostFix}\ \langle (\texttt{postfix}\ N_{numargs}\ Q_{body}), N_{inputs}^*\rangle)).
\end{aligned}$$

This completes the sketch of the proof that the denotational semantics for PostFix is sound with respect to the operational semantics for PostFix. The fact that all PostFix programs terminate simplifies the proof, because it is not necessary to consider the case of infinitely long transition paths (in which case $(beh_{det}\ \langle P, I\rangle) = \infty$). For languages containing nonterminating programs, a denotational soundness proof must also explicitly handle this case.

▷ **Exercise 4.12** Complete the proof that the denotational semantics for PostFix is sound with respect to its operational semantics by fleshing out the following details:

a. Show that lemma 2 holds for each transition rule in Figure 3.4.

b. Make a list of all stuck configuration patterns in the PostFix SOS and show that lemma 3 holds for each such pattern. ◁

▷ **Exercise 4.13** Show that the denotational semantics for each of the following languages is sound with respect to its operational semantics: (1) a version of ELMM whose operators include only +, -, and *; (2) full ELMM; (3) ELM; and (4) EL. ◁

4.4.4.2 Adequacy

The notion of soundness developed above works at the level of a whole program. But often we want to reason about smaller phrases within a program. In particular, we want to reason that we can substitute one phrase for another without changing the operational behavior of the program. The following adequacy property says that denotational equivalence implies the operational notion of observational equivalence:

> **Adequacy:** Suppose that $\mathbb{P}$ ranges over program contexts, H ranges over the kinds of phrases that fill the holes in program contexts, and $\mathcal{H}$ is a denotational meaning function for phrases. A denotational semantics is **adequate with respect to (wrt)** an operational semantics if the following holds:
>
> $$\mathcal{H}[\![H_1]\!] = \mathcal{H}[\![H_2]\!] \text{ implies } H_1 =_{obs} H_2.$$

Recall from page 84 that $H_1 =_{obs} H_2$ means that for all program contexts $\mathbb{P}$ and all inputs I, $beh\ \langle \mathbb{P}\{H_1\}, I\rangle = beh\ \langle \mathbb{P}\{H_2\}, I\rangle$

In the case of a deterministic behavior function, the following reasoning shows that adequacy is *almost* implied by denotational soundness:

$\mathcal{H}[\![H_1]\!] = \mathcal{H}[\![H_2]\!]$

implies $\mathcal{P}[\![\mathbb{P}\{H_1\}]\!] = \mathcal{P}[\![\mathbb{P}\{H_2\}]\!]$, by compositionality of denotational semantics

implies $\mathit{meaning}\ \langle \mathbb{P}\{H_1\}, (\mathit{in}\ I)\rangle = \mathit{meaning}\ \langle \mathbb{P}\{H_2\}, (\mathit{in}\ I)\rangle$ for any inputs I

implies $(\mathit{out}\ (\mathit{beh}_{\mathit{det}}\ \langle \mathbb{P}\{H_1\}, I\rangle)) = (\mathit{out}\ (\mathit{beh}_{\mathit{det}}\ \langle \mathbb{P}\{H_2\}, I\rangle))$, by soundness

But demonstrating the observational equivalence requires showing

$$\mathit{beh}_{\mathit{det}}\ \langle \mathbb{P}\{H_1\}, I\rangle = \mathit{beh}_{\mathit{det}}\ \langle \mathbb{P}\{H_2\}, I\rangle.$$

To conclude this from the above line of reasoning requires an additional property. Suppose that A ranges over observable answer expressions in the syntactic domain Answer. Then we need a property we shall call **denotational distinctness of observables**:

$$(\mathit{out}\ A_1) = (\mathit{out}\ A_2) \text{ implies } A_1 = A_2.$$

Recall that *out* maps syntactic answers to semantic ones. So the above property requires that syntactically distinct answers be denotationally distinct. That is, we cannot have two observationally distinct answers answers with the same meaning.

Both EL and PostFix have denotational distinctness of observables. In each language, observable answers are either integer numerals or an error token. Assuming that only canonical integer numerals are used (e.g., `17` rather than `017` or `+17`) distinct integer numerals denote distinct integers. Note that PostFix would *not* have this property if executable sequences at the top of a final stack could be returned as observable answers. For example, the syntactically distinct sequences `(1 add 2 add)` and `(3 add)` would both denote the same transformation:

$$(\mathit{push}\ (\mathit{Value} \mapsto \mathit{Result}\ (\mathit{Int} \mapsto \mathit{Value}\ 3))).$$

The above discussion allows us to conclude that any language with denotational soundness and denotational distinctness of observables has the adequacy property. In turn, this property justifies the use of denotational reasoning for proving the safety of program transformations. For example, the PostFix Denotational Equivalence Theorem on page 141 is a corollary of the adequacy of PostFix.

4.4.4.3 Full Abstraction

Changing the unidirectional implication of adequacy to a bidirectional implication yields a stronger property called **full abstraction**:

> **Full Abstraction:** Suppose that $\mathbb{P}$ ranges over program contexts, H ranges over the kinds of phrases that fill the holes in program contexts, and $\mathcal{H}$ is a denotational meaning function for phrases. A denotational semantics is **adequate with respect to (wrt)** an operational semantics if the following holds:
>
> $$\mathcal{H}[\![H_1]\!] = \mathcal{H}[\![H_2]\!] \text{ iff } H_1 =_{obs} H_2.$$

In addition to adequacy, full abstraction requires that observational equivalence imply denotational equivalence. That is, program fragments that behave the same in all contexts must have the same denotational meaning.

The various dialects of EL we have considered are all fully abstract. Consider the restricted version of ELMM in which the only operations are `+`, `-`, and `*`. In this language, every numerical expression denotes an integer. We already know that this language is adequate; to prove full abstraction, we need to show that observational equivalence implies denotational equivalence. We will prove this by contradiction. Suppose that $NE_1 =_{obs} NE_2$, but $\mathcal{NE}[\![NE_1]\!] \neq \mathcal{NE}[\![NE_2]\!]$. Consider the ELMM context $\mathbb{P} =$ `(elmm` $\Box$`)`. Modeling the non-existent inputs in this case by *unit*, we have:

$(out\ \ (beh_{det}\ \langle \mathbb{P}\{NE_1\}, unit\rangle))$
$= \mathcal{P}[\![\mathbb{P}\{NE_1\}]\!]$, by soundness
$= \mathcal{NE}[\![NE_1]\!]$, by definition of $\mathcal{P}$
$\neq \mathcal{NE}[\![NE_2]\!]$, by assumption
$= \mathcal{P}[\![\mathbb{P}\{NE_2\}]\!]$, by definition of $\mathcal{P}$
$= (out\ \ (beh_{det}\ \langle \mathbb{P}\{NE_2\}, unit\rangle))$, by soundness

Because ELMM has denotational distinctness of observables, we conclude that $NE_1 \neq_{obs} NE_2$, contradicting our original assumption. A similar proof works to show full abstraction for the other dialects of EL we have studied.

Surprisingly, PostFix is *not* fully abstract! As argued in Section 4.4.3, even though all PostFix programs terminate, the denotational domains for answers and stacks in PostFix must include an entity denoting nontermination, which we will write as $\perp$. This is the denotational analog of the operational token ∞. Even though no PostFix command sequence can loop, the presence of $\perp$ in the semantics can distinguish the meanings of some observationally equivalent command sequences.

For example, consider the following two command sequences:

$Q_1 =$ `1 0 div`
$Q_2 =$ `exec 1 0 div`.

Q_1 signals an error for any stack. Q_2 first executes the top value V_{top} on the stack and then executes `1 0 div`. We argue that Q_2 is observationally equivalent to Q_1, because it will also signal an error for any stack:

- if the stack is empty or if V_{top} is not an executable sequence, the attempt to perform `exec` will fail with an error;
- if V_{top} is an executable sequence, Q_2 will execute it. Since all POSTFIX command sequences terminate, the execution of V_{top} will either signal an error, or it will terminate without an error. In the latter case, the execution continues with `1 0 div`, which necessarily signals an error.

Even though $Q_1 =_{obs} Q_2$, they do not denote the same stack transform! To see this, consider a stack transform $t_{weird} = \lambda s \,.\, \bot$ and a stack s_{weird} whose top value is $(\mathit{StackTransform} \mapsto \mathit{Value}\ t_{weird})$. Both t_{weird} and s_{weird} are "weird" in the sense that they can never arise during a POSTFIX computation, in which all stack transforms necessarily terminate. Nevertheless, t_{weird} is a legal element of the domain *StackTransform*, and it must be considered as a legal stack element in denotational reasoning. Observe that $(\mathcal{Q}[\![Q_1]\!]\ s_{weird}) = \mathit{errorStack}$, but $(\mathcal{Q}[\![Q_2]\!]\ s_{weird}) = \bot$ — i.e., the latter computation does not terminate. So Q_1 and Q_2 denote distinct stack transforms even though they are observationally equivalent.

Intuitively, full abstraction says that the semantic domains don't contain any extra "junk" that can't be expressed by phrases in the language. In the case of POSTFIX, the domains harbor $\bot$ even though it cannot be expressed in the language.

4.4.5 Operational vs. Denotational: A Comparison

We have noted in this chapter 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, it's 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.

Reading

Denotational semantics was invented by Christopher Strachey and Dana Scott. For a tutorial introduction to denotational semantics, we recommend the articles [Ten76] and [Mos90]. Coverage of both operational and denotational semantics along with their use in reasoning about several simple programming languages can be found in several semantics textbooks [Gun92, Win93, Mit96]. Full-length books devoted to denotational semantics include [Gor79, Sto85, Sch86a].

Our notions of denotational soundness and adequacy are somewhat different than (but related to) those in the literature. For a discussion of (the traditional approach to) soundness, adequacy, and full abstraction, see [Gun92].