Chapter 3

Operational Semantics

And now I see with eye serene
The very pulse of the machine.

— She Was a Phantom of Delight, William Wordsworth

3.1 The Operational Semantics Game

Consider executing the following PostFix program on the arguments [4, 5]:

```
(postfix 2 (2 (3 mul add) exec) 1 swap exec sub)
```

It helps to have a bookkeeping notation that represents the process of applying the informal rules presented in Chapter 1. For example, the table in Figure 3.1 illustrates one way to represent the execution of the above program. The table has two columns: the first column in each row holds the current command sequence; the second holds the current stack. The execution process begins by filling the first row of the table with the command sequence of the given program and an empty stack. Execution proceeds in a step-by-step fashion by using the rule for the first command of the current row to generate the next row. Each execution step removes the first command from the sequence and updates the stack. In the case of `exec`, new commands may also be prepended to the command sequence. The execution process terminates as soon as a row with an empty command sequence is generated. The result of the execution is the top stack element of the final row (`-3` in the example).

The table-based technique for executing PostFix programs exemplifies an **operational semantics**. Operational semantics formalizes the common intuition that program execution can be understood as a step-by-step process that

Commands	Stack
`(2 (3 mul add) exec) 1 swap exec sub`	4 5
`1 swap exec sub`	`(2 (3 mul add) exec)` 4 5
`swap exec sub`	1 `(2 (3 mul add) exec)` 4 5
`exec sub`	`(2 (3 mul add) exec)` 1 4 5
`2 (3 mul add) exec sub`	1 4 5
`(3 mul add) exec sub`	2 1 4 5
`exec sub`	`(3 mul add)` 2 1 4 5
`3 mul add sub`	2 1 4 5
`mul add sub`	3 2 1 4 5
`add sub`	6 1 4 5
`sub`	7 4 5
	-3 5

Figure 3.1: A table showing the step-by-step execution of a PostFix program.

evolves by the mechanical application of a fixed set of rules. Sometimes the rules describe how the state of some physical machine is changed by executing an instruction. For example, assembly code instructions are defined in terms of the effect that they have on the architectural elements of a computer: registers, stack, memory, instruction stream, etc. But the rules may also describe how language constructs affect the state of some **abstract machine** that provides a mathematical model for program execution. Each state of the abstract machine is called a **configuration**.

For example, in the PostFix abstract machine implied by the table in Figure 3.1, each configuration is modeled by one row of the execution table: a pair of a program and a stack. The next configuration of the machine is determined from the current one based on the first command in the current program. The behavior of each command can be specified in terms of how it transforms the current configuration into the next one. For example, executing the `add` command removes it from the command sequence and replaces the top two elements of the stack by their sum. Executing the `exec` command pops an executable sequence from the top of the stack and prepends its commands in front of the commands following `exec`.

The general structure of an operational semantics execution is illustrated in Figure 3.2. An abstract machine accepts a program to be executed along with its inputs and then chugs away until it emits an answer. Internally, the abstract machine typically manipulates configurations with two kinds of parts:

1. The **code component**: a program phrase that controls the rest of the computation.

2. The **state components**: entities that are manipulated by the program during its execution. In the case of PostFix, the single state component is a stack, but configurations for other languages might include state components modeling random-access memory, a set of name/object bindings, a file system, a graphics state, various kinds of control information, etc. Sometimes there are no state components, in which case a configuration is just code.

The stages of the operational execution are as follows:

- The program and its inputs are first mapped by an **input function** into an **initial configuration** of the abstract machine. The code component of the initial configuration is usually some part of the given program, and the state components are appropriately initialized from the inputs. For instance, in an initial configuration for PostFix, the code component is the

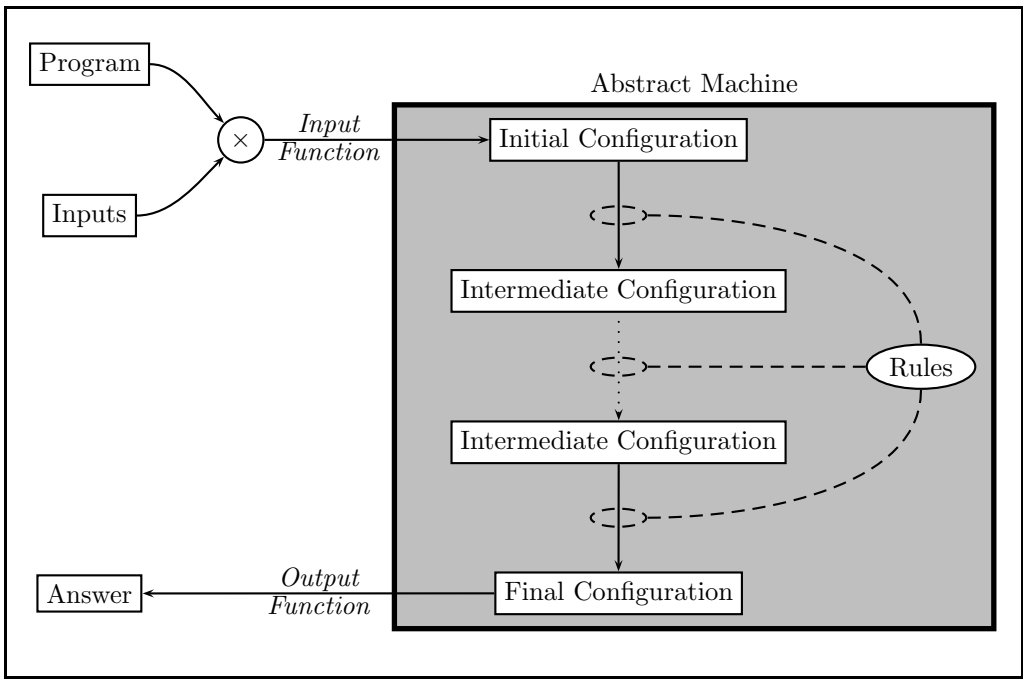


Figure 3.2: The operational semantics "game board."

command sequence body of the program and the single state component is a stack containing the integer arguments in order with the first argument at the top of the stack.

- After an initial configuration has been constructed, it's time to "turn the crank" of the abstract machine. During this phase, the rules governing the abstract machine are applied in an iterative fashion to yield a sequence of intermediate configurations. Each configuration is the result of one step in the step-by-step execution of the program. This stage continues until a configuration is reached that is deemed to be a **final configuration**. What counts as a final configuration varies widely between abstract machines. In the case of PostFix, a configuration is final when the code component is an empty command sequence.

- The last step of execution is mapping the final configuration to an **answer** via an **output function**. What is considered to be an answer differs greatly from language to language. For PostFix, the answer is the top stack value in a final configuration, if it's an integer. If the stack is empty or the top value is an executable sequence, the answer is an error token. In

other systems, the answer might also include elements like the final state of the memory, file system, or graphics screen.

Sometimes an abstract machine never reaches a final configuration. This can happen for one of two reasons:

1. The abstract machine may reach a non-final configuration to which no rules apply. Such a configuration is said to be a **stuck state**. For example, the initial configuration for the PostFix program `(postfix 1 sub)` is a stuck state because the rules in Figure 1.1 don't say how to handle `sub` when the stack doesn't contain at least two elements. (The configuration is not final because the command sequence [`sub`] is non-empty.) Stuck states often model error situations.

2. The rule-applying process of the abstract machine might not terminate. In any universal programming language[1] it is possible to write programs that loop forever. For such programs, the execution process of the abstract machine never terminates. As a consequence of the halting theorem[2], we can't do better than this: there's no general way to tweak the abstract machine of a universal language so that it always indicates when it is in an infinite loop.

We show in Section 3.4.3 that all PostFix programs must terminate. This implies that PostFix is not universal.

3.2 Small-step Operational Semantics (SOS)

3.2.1 Formal Framework

Above, we presented a high-level introduction to operational semantics. Here, we iron out all the details necessary to turn this approach into a formal framework known as **small-step operational semantics** (SOS[3]). An SOS is characterized by the use of **rewrite rules** to specify the step-by-step transformation of configurations in an abstract machine.

To express this framework formally, we will use the mathematical metalanguage described in Appendix A. Before reading further, you should at least

[1] A programming language is **universal** if it can express all computable functions.

[2] The **halting theorem** states that there is no program that can decide for all programs P and all inputs A whether P terminates on A.

[3] This framework, due to Plotkin [Plo81], was originally called **structured operational semantics**. It later became known as the small-step approach to distinguish it from – you guessed it – a big-step approach (see Section 3.3).

skim this appendix to familiarize yourself with the notational conventions of the metalanguage. Later, when you encounter an unfamiliar notation or concept, consult the relevant section of the appendix for a detailed explanation.

Consider a programming language L with legal programs $P \in$ Program, inputs $I \in$ Inputs, and elements $A \in$ Answer that are considered to be valid answers to programs. Then an SOS for L is a five-tuple $SOS = \langle CF, \Rightarrow, FC, IF, OF \rangle$, where:

- CF is the set of **configurations** for an abstract machine for L. The metavariable cf ranges over configurations.

- $\Rightarrow$, the **transition relation**, is a binary relation between configurations that defines the allowable transitions between configurations. The notation $cf \Rightarrow cf'$ means that there is a **(one step) transition** from the configuration cf to the configuration cf'. This notation, which is shorthand for $\langle cf, cf' \rangle \in \Rightarrow$, is pronounced "$cf$ rewrites to cf' in one step." The two parts of a transition have names: cf is called the **left hand side (LHS)** and cf' is called the **right hand side (RHS)**. The transition relation is usually specified by rewrite rules, as described below in Section 3.2.3.

  The reflexive, transitive closure of $\Rightarrow$ is written $\stackrel{*}{\Rightarrow}$. So $cf \stackrel{*}{\Rightarrow} cf'$ means that cf rewrites to cf' in zero or more steps. The sequence of transitions between cf and cf' is called a **transition path**. The **length** of a transition path is the number of transitions in the path. The notation $cf \stackrel{n}{\Rightarrow} cf'$ indicates that cf rewrites to cf' in n steps, i.e., via a transition path of length n. The notation $cf \stackrel{\infty}{\Rightarrow}$ indicates that there is an infinitely long transition path beginning with cf.

  A configuration cf is **reducible** if there is some cf' such that $cf \Rightarrow cf'$. If there is no such cf', then we write $cf \not\Rightarrow$ and say that cf is **irreducible**. CF can be partitioned into two sets, $Reducible_{SOS}$ (containing all reducible configurations) and $Irreducible_{SOS}$ (containing all irreducible ones). We omit the SOS subscript when it is clear from context. A transition relation $\Rightarrow$ is **deterministic** if for every $cf \in Reducible_{SOS}$ there is exactly one cf' such that $cf \Rightarrow cf'$. Otherwise, $\Rightarrow$ is said to be **non-deterministic**.

- FC, the set of **final configurations**, is a subset of $Irreducible_{SOS}$ containing all configurations that are considered to be final states in the execution of a program. The set $Stuck_{SOS}$ of stuck states is defined to be $(Irreducible_{SOS} - FC)$ — i.e., the non-final irreducible configurations. We omit the SOS subscript when it is clear from context.

- IF : Program $\times$ Inputs $\rightarrow$ CF is an **input function** that maps a program and its inputs into an initial configuration.

- OF : $FC \rightarrow$ Answer is an **output function** that maps a final configuration to an appropriate answer domain.

An SOS defines the behavior of a program in a way that we shall now make precise. What are the possible behaviors of a program? As discussed above, a program either (1) returns an answer (2) gets stuck in a non-final irreducible configuration or (3) loops infinitely. We model these via the following Outcome domain, where `stuckout` designates a stuck program and `loopout` designates a infinitely looping program:

$$
\begin{array}{l}
\quad \text{StuckOut} = \{\texttt{stuckout}\} \\
\quad \text{LoopOut} = \{\texttt{loopout}\} \\
o \in \text{Outcome} = \text{Answer} + \text{StuckOut} + \text{LoopOut} \\
\quad \texttt{stuck} = (\text{StuckOut} \mapsto \text{Outcome } \texttt{stuckout}) \\
\quad \infty = (\text{StuckOut} \mapsto \text{Outcome } \texttt{loopout})
\end{array}
$$

Suppose that an SOS has a deterministic transition relation. Then we can define the behavior of a program P on inputs I as follows:

$$
\begin{array}{l}
beh_{det} : \text{Program} \times \text{Inputs} \rightarrow \text{Outcome} \\
beh_{det}\ \langle P, I \rangle = \begin{cases} (\text{Answer} \mapsto \text{Outcome}\ \ (OF\ \ cf)) & \text{if } (IF\ \ \langle P, I \rangle) \stackrel{*}{\Rightarrow} cf \in FC \\ \texttt{stuck} & \text{if } (IF\ \ \langle P, I \rangle) \stackrel{*}{\Rightarrow} cf \in Stuck \\ \infty & \text{if } (IF\ \ \langle P, I \rangle) \stackrel{\infty}{\Rightarrow} \end{cases}
\end{array}
$$

In the first case, an execution starting at the initial configuration eventually reaches a final configuration, whose answer is returned. In the second case, an execution starting at the initial configuration eventually gets stuck at a non-final configuration. In the last case, there is an infinite transition path starting at the initial configuration, so the program never halts.

What if the transition relation is not deterministic? In this case, it is possible that there are multiple transition paths starting at the initial configuration. Some of these might end at final configurations with different answers. Others might be infinitely long or end at stuck states. In general, we must allow for the possibility that there are many outcomes, so the signature of the behavior function *beh* in this case must return a *set* of outcomes — i.e., an element of the

powerset domain $\mathcal{P}(\text{Outcome})$[4]

$$beh : \text{Program} \times \text{Inputs} \to \mathcal{P}(\text{Outcome})$$

$$o \in (beh\ \langle P, I\rangle) \text{ if } \begin{cases} o = (\text{Answer} \mapsto \text{Outcome}\ \ (OF\ \ cf)) \\ \quad \text{and } (IF\ \ \langle P, I\rangle) \stackrel{*}{\Rightarrow} cf \in FC \\ o = \texttt{stuck} \text{ and } (IF\ \ \langle P, I\rangle) \stackrel{*}{\Rightarrow} cf \in \mathit{Stuck} \\ o = \infty \text{ and } (IF\ \ \langle P, I\rangle) \stackrel{\infty}{\Rightarrow} \end{cases}$$

An SOS with a non-deterministic transition relation won't necessarily give rise to results that contain multiple outcomes. Indeed, we will see later (in Section 3.4.2) that some systems with non-deterministic transition relations can still have a behavior that is deterministic — i.e., the resulting set of outcomes is always a singleton.

3.2.2 Example: An SOS for PostFix

We can now formalize the elements of the PostFix SOS described informally in Section 3.1 (except for the transition relation, which will be formalized in Section 3.2.3). The details are presented in Figure 3.3. A stack is a sequence of values that are either integer numerals (from domain Intlit) or executable sequences (from domain Commands). PostFix programs take a sequence of integer numerals as their inputs, and, when no error is encountered, return an integer numeral as an answer. A configuration is a pair of a command sequence and a stack. A final configuration is one whose command sequence is empty and whose stack is non-empty with an integer numeral on top (i.e., an element of FinalStack). The input function *IF* maps a program and its numeric inputs to a configuration consisting of the body command sequence and an initial stack with the inputs arranged from top down. If the number of arguments N expected by the program does not match the actual number n of arguments supplied, then *IF* returns a stuck configuration $\langle []_{\text{Command}}, []_{\text{Value}}\rangle$ that represents an error. The output function *OF* returns the top integer numeral from stack of a final configuration.

The PostFix SOS in Figure 3.3 models errors using stuck states. By definition, stuck states are exactly those irreducible configurations that are non-final. In PostFix, stuck states are irreducible configurations whose command sequence is non-empty or those that pair an empty command sequence with a stack that is empty or has an executable sequence on top. The outcome of a program that reaches such a configuration will be `stuck`.

[4]The result of *beh* must in fact be a *non-empty* set of outcomes, since every program will have at least one outcome.

Domains

$V \in \text{Value} = \text{Intlit} + \text{Commands}$
$S \in \text{Stack} = \text{Value}^*$
$\text{FinalStack} = \{S \mid (\mathit{length}\ S) \geq 1 \text{ and } (\mathit{nth}\ 1\ S) = (\text{Intlit} \mapsto \text{Value}\ N)\}$
$\text{Inputs} = \text{Intlit}^*$
$\text{Answer} = \text{Intlit}$

SOS

Suppose that the PostFix SOS has the form $PFSOS = \langle CF, \Rightarrow, FC, IF, OF\rangle$. Then the SOS components are:

$CF = \text{Commands} \times \text{Stack}$

$\Rightarrow$ is a deterministic transition relation defined in Section 3.2.3

$FC = \{[\,]_{\text{Command}}\} \times \text{FinalStack}$

$IF : \text{Program} \times \text{Inputs} \to CF$
$= \lambda\langle \texttt{(postfix}\ N\ Q\texttt{)}, [N_1, \ldots, N_n]\rangle\ .$
$\quad$ **if** $N = \text{n}$ **then** $\langle Q, [(\text{Intlit} \mapsto \text{Value}\ N_1), \ldots, (\text{Intlit} \mapsto \text{Value}\ N_n)]\rangle$
$\quad$ **else** $\langle [\,]_{\text{Command}}, [\,]_{\text{Value}}\rangle$ **fi**

$OF : FC \to \text{Answer} = \lambda\langle [\,]_{\text{Command}}, (\text{Intlit} \mapsto \text{Value}\ N)\ .\ S'\rangle\ .\ N$

Figure 3.3: An SOS for PostFix.

Although it is convenient to use stuck states to model errors, it is not strictly necessary. With some extra work, it is always possible to modify the final configuration set FC and the output function OF so that such programs instead have as their meaning some error token in Answer. Using PostFix as an example, we can use a modified answer domain Answer′ that includes an error token, a modified final configuration set FC' that includes all irreducible configurations, and the modified OF' shown below:

$\text{Error} = \{\texttt{error}\}$
$\text{Answer}' = \text{Intlit} + \text{Error}$

$FC' = \mathit{Irreducible}_{PFSOS}$

$OF : FC' \to \text{Answer}'$
$= \lambda\langle Q, V^*\rangle\ .$ **matching** $\langle Q, V^*\rangle$
$\quad \triangleright \langle [\,]_{\text{Command}}, (\text{Intlit} \mapsto \text{Value}\ N)\ .\ S'\rangle \ [\!] \ (\text{Intlit} \mapsto \text{Answer}'\ N)$
$\quad \triangleright$ **else** $(\text{Error} \mapsto \text{Answer}'\ \texttt{error})$

With these modifications, the behavior of a PostFix program that encounters an error will be $(\text{Answer}' \mapsto \text{Outcome}\ (\text{Error} \mapsto \text{Answer}'\ \texttt{error}))$ rather than `stuck`.

▷ **Exercise 3.1** Look up definitions of the following kinds of automata and express each of them in the SOS framework: deterministic finite automata, non-deterministic finite automata, deterministic pushdown automata, and Turing machines. Represent strings, stacks, and tapes as sequences of symbols. ◁

3.2.3 Rewrite Rules

The transition relation, $\Rightarrow$, for an SOS is often specified by a set of **rewrite rules**. A rewrite rule has the form

$$\frac{\textit{antecedents}}{\textit{consequent}} \qquad [\textit{rule-name}]$$

where the antecedents and the consequent contain transition patterns (described below). Informally, the rule asserts: "If the transitions specified by the *antecedents* are valid, then the transition specified by the consequent is valid." The label [*rule-name*] on the rule is just a handy name for referring to the rule, and is not a part of the rule structure. A rewrite rule with no antecedents is an **axiom**; otherwise it is a **progress rule**. The horizontal bar is usually omitted when writing an axiom.

A complete set of rewrite rules for PostFix appears in Figure 3.4. All of the rules are axioms. Together with the definitions of CF, FC, IF, and OF, these rules constitute a formal SOS version of the informal PostFix semantics originally presented in Figure 1.1. We will spend the rest of this section studying the meaning of these rules and considering alternative rules.

3.2.3.1 Axioms

Since an axiom has no antecedents, it is determined solely by its consequent. As noted above, the consequent must be a **transition pattern**. A transition pattern looks like a transition except that the LHS and RHS may contain domain variables interspersed with the usual notation for configurations. Informally, a transition pattern is a schema that stands for all the transitions that match the pattern. An axiom stands for the collection of all configuration pairs that match the LHS and RHS of the transition pattern, respectively.

As an example, let's consider in detail the axiom that defines the behavior of PostFix numerals:

$$\langle N \ . \ Q, \ S\rangle \Rightarrow \langle Q, \ N \ . \ S\rangle \qquad [\textit{num}]$$

This axiom stands for an infinite number of pairs of configurations of the form $\langle cf, cf'\rangle$. It says that if cf is a configuration in which the command sequence is

$$\langle N \ . \ Q, \ S\rangle \Rightarrow \langle Q, \ N \ . \ S\rangle \qquad [num]$$

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

$$\langle \texttt{pop} \ . \ Q, \ V_{top} \ . \ S\rangle \Rightarrow \langle Q, \ S\rangle \qquad [pop]$$

$$\langle \texttt{swap} \ . \ Q, \ V_1 \ . \ V_2 \ . \ S\rangle \Rightarrow \langle Q, \ V_2 \ . \ V_1 \ . \ S\rangle \qquad [swap]$$

$$\langle \texttt{sel} \ . \ Q_{rest}, \ V_{false} \ . \ V_{true} \ . \ \texttt{0} \ . \ S\rangle \Rightarrow \langle Q_{rest}, \ V_{false} \ . \ S\rangle \qquad [sel\text{-}false]$$

$$\langle \texttt{sel} \ . \ Q_{rest}, \ V_{false} \ . \ V_{true} \ . \ N_{test} \ . \ S\rangle \Rightarrow \langle Q_{rest}, \ V_{true} \ . \ S\rangle, \quad \text{where } N_{test} \neq \texttt{0} \qquad [sel\text{-}true]$$

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

$$\langle A \ . \ Q, \ N_1 \ . \ N_2 \ . \ S\rangle \Rightarrow \langle Q, \ N_{result} \ . \ S\rangle, \quad \text{where } N_{result} = (calculate \ A \ N_2 \ N_1) \qquad [arithop]$$

$$\langle R \ . \ Q, \ N_1 \ . \ N_2 \ . \ S\rangle \Rightarrow \langle Q, \ \texttt{1} \ . \ S\rangle, \quad \text{where } (compare \ R \ N_2 \ N_1) \qquad [relop\text{-}true]$$

$$\langle R \ . \ Q, \ N_1 \ . \ N_2 \ . \ S\rangle \Rightarrow \langle Q, \ \texttt{0} \ . \ S\rangle, \quad \text{where } \neg\,(compare \ R \ N_2 \ N_1) \qquad [relop\text{-}false]$$

$$\langle \texttt{nget} \ . \ Q, \ N_{index} \ . \ [V_1, \ldots, V_{N_{size}}]\rangle \Rightarrow \langle Q, \ V_{N_{index}} \ . \ [V_1, \ldots, V_{N_{size}}]\rangle, \quad \text{where } (compare \ \texttt{gt} \ N_{index} \ \texttt{0}) \wedge \neg\,(compare \ \texttt{gt} \ N_{index} \ N_{size}) \qquad [nget]$$

Figure 3.4: Rewrite rules defining the transition relation ($\Rightarrow$) for PostFix.

a numeral N followed by Q and the stack is S, then there is a transition from cf to a configuration cf' whose command sequence is Q, and whose stack holds N followed by S.

In the [*num*] rule, N, Q, and S are domain variables that act as patterns that can match any element in the domain over which the variable ranges. Thus, N matches any integer numeral, Q matches any command sequence, and S matches any stack. When the same pattern variable occurs more than once within a rule, all occurrences must denote the same element; this constrains the class of transitions specified by the rule. Thus, the [*num*] rule matches the transition

$$\langle \texttt{(17 add swap)},\ [\texttt{19},\ \texttt{(2 mul)}]\rangle \Rightarrow \langle \texttt{(add swap)},\ [\texttt{17},\ \texttt{19},\ \texttt{(2 mul)}]\rangle$$

with $N = \texttt{17}$, $Q = [\texttt{add},\ \texttt{swap}]$, and $S = [\texttt{19},\ [\texttt{2},\ \texttt{mul}]]$. On the other hand, the rule does not match the transition

$$\langle \texttt{(17 add swap)},\ [\texttt{19},\ \texttt{(2 mul)}]\rangle \Rightarrow \langle \texttt{(add swap)},\ [\texttt{17},\ \texttt{19},\ \texttt{(2 mul)},\ \texttt{23}]\rangle$$

because there is no consistent interpretation for the pattern variable S — it is $[\texttt{19},\ [\texttt{(2 mul)}]]$ in the LHS of the transition, and $[\texttt{19},\ \texttt{(2 mul)},\ \texttt{23}]$ in the RHS.

As another example, the configuration pattern $\langle Q,\ N\ .\ N\ .\ S\rangle$ would only match configurations with stacks in which the top two values are the same integer numeral. If the RHS of the [*num*] rule consequent were replaced with this configuration pattern, then the rule would indicate that two copies of the integer numeral should be pushed onto the stack.

At this point, the meticulous reader may have noticed that in the rewrite rules and sample transitions we have taken many liberties with our notation. If we had strictly adhered to our metalanguage notation, then we would have written the [*num*] rule as

$$\langle (\text{Intlit} \mapsto \text{Command}\ N)\ .\ Q, S\rangle \Rightarrow \langle Q, (\text{Intlit} \mapsto \text{Value}\ N)\ .\ S\rangle \qquad [num]$$

and we would have written the matching transition as

$$\begin{aligned}
&\langle [\texttt{17}, \texttt{add}, \texttt{swap}]_{\text{Command}},\ [(\text{Intlit} \mapsto \text{Value}\ \texttt{19}),\\
&\qquad\qquad (\text{Commands} \mapsto \text{Value}\ [\texttt{2}, \texttt{mul}]_{\text{Command}})\ ]\rangle\\
&\Rightarrow \langle [\texttt{add}, \texttt{swap}]_{\text{Command}},\ [(\text{Intlit} \mapsto \text{Value}\ \texttt{17}),\\
&\qquad\qquad (\text{Intlit} \mapsto \text{Value}\ \texttt{19}),\\
&\qquad\qquad (\text{Commands} \mapsto \text{Value}\ [\texttt{2}, \texttt{mul}]_{\text{Command}})\ ]\rangle.
\end{aligned}$$

However, we believe that the more rigorous notation severely impedes the readability of the rules and examples. For this reason, we will stick with our stylized notation when it is unlikely to cause confusion. In particular, in operational semantics rules and sample transitions, we adopt the following conventions:

- Injections will be elided when they are clear from context. For example, if N appears as a command, then it stands for (Intlit $\mapsto$ Command N), while if it appears as a stack element, then it stands for (Intlit $\mapsto$ Value N).

- Sequences of syntactic elements will often be written as parenthesized s-expressions. For example, the PostFix command sequence

$$[\texttt{3},\ [\texttt{2},\ \texttt{mul}]_{\text{Command}},\ \texttt{swap}]_{\text{Command}}$$

  will be abbreviated as

```
(3 (2 mul) swap).
```

  The former is more precise, but the latter is easier to read. In PostFix examples, we have chosen to keep the sequence notation for stacks to visually distinguish the two components of a configuration.

Despite these notational acrobatics, keep in mind that we are manipulating well-defined mathematical structures. So it is always possible to add the appropriate decorations to make the notation completely rigorous.[5]

Some of the PostFix rules ([*arithop*], [*relop-true*], [*relop-false*], [*sel-true*], and [*nget*]) include **side conditions** that specify additional restrictions on the domain variables. For example, consider the axiom which handles a conditional whose test is true:

$$\langle \texttt{sel}\ .\ Q_{rest},\ \ V_{false}\ .\ V_{true}\ .\ N_{test}\ .\ S\rangle \Rightarrow \langle Q_{rest},\ \ V_{true}\ .\ S\rangle, \quad \text{where } N_{test} \neq \texttt{0} \qquad [\textit{sel-true}]$$

This axiom encodes the fact that `sel` treats any nonzero integer numeral as true. It says that as long as the test numeral N_{test} (the third element on the stack) is not the same syntactic object as `0`, then the next configuration is obtained by removing `sel` from the command sequence, and pushing the second stack element on the result of popping the top three elements off of the stack. The domain variable N_{test} that appears in the side condition $N_{test} \neq \texttt{0}$ stands for the same entity that N_{test} denotes in the LHS of the consequent, providing the link between the transition pattern and the side condition. Note how the domain variables and the structure of the components are used to constrain the pairs of configurations that satisfy this rule. This rule only represents pairs $\langle cf, cf'\rangle$ in which the stack of cf contains at least three elements, the third of which is a nonzero integer numeral. The rule does not apply to configurations whose stacks have fewer than three elements, or whose third element is an executable sequence or the numeral `0`.

[5] But those who pay too much attention to rigor may develop rigor mortis!

The side conditions in the [*arithop*], [*relop*], and [*nget*] rules deserve some explanation. The *calculate* function used in the side condition of [*arithop*] returns the numeral N_{result} resulting from the application of the operator A to the operands N_2 and N_1; it abstracts away the details of such computations.[6] We assume that *calculate* is a partial function that is undefined when A is `div` and N_1 is `0`, so division by zero yields a stuck state. The [*relop-true*] and [*relop-false*] rules are similar to [*arithop*]; here the auxiliary *compare* function is assumed to return the truth value resulting from the associated comparison. The rules then convert this truth value into a PostFix value of `1` (true) or `0` (false). In the [*nget*] rule, the *compare* function is used to ensure that the numeral N_{index} is a valid index for one of the values on the stack. If not, the configuration is stuck. In the side conditions, the symbol $\neg$ stands for logical negation and $\wedge$ stands for logical conjunction.

You should now know enough about the rule notation to understand all of the rewrite rules in Figure 3.4. The [*num*] and [*seq*] rules push the two different kinds of values onto the stack. The [*swap*], [*pop*], [*sel-true*], and [*sel-false*] rules all perform straightforward stack manipulations. The [*exec*] rule prepends an executable sequence from the stack onto the command sequence following the current command.

It is easy to see that the transition relation defined in Figure 3.4 is deterministic. The first command in the command sequence of a configuration uniquely determines which transition pattern might match, except for the case of `sel`, where the third stack value distinguishes whether [*sel-true*] or [*sel-false*] matches. The LHS of each transition pattern can match a given configuration in at most one way. So for any given PostFix configuration cf, there is at most one cf' such that $cf \Rightarrow cf'$.

3.2.3.2 Operational Execution

The operational semantics can be used to execute a PostFix program in a way similar to the table-based method presented earlier. For example, the execution of the PostFix program shown earlier in Figure 3.1 is illustrated in Figure 3.5. The input function is applied to the program to yield an initial configuration, and then a series of transitions specified by the rewrite rules are applied. In the

[6] Note that *calculate* manipulates *numerals* (i.e., names for integers) rather than the *integers* that they name. This may seem pedantic, but we haven't described yet how the meaning of an integer numeral is determined. In fact, integers are never even used in the SOS for PostFix. If we had instead defined the syntax of PostFix to use integers rather than integer numerals, then we could have used the usual integer addition operation here. But we chose integer numerals to emphasize the syntactic nature of operational semantics.

figure, the configuration resulting from each transition appears on a separate line and is labeled by the applied rule. When a final configuration is reached, the output function is applied to this configuration to yield `-3`, which is the result computed by the program. We can summarize the transition path from the initial to the final configuration as

$$\langle \texttt{((2 (3 mul add) exec) 1 swap exec sub)}, [\texttt{4}, \texttt{5}]\rangle \stackrel{10}{\Longrightarrow} \langle \texttt{()}, [\texttt{-3}, \texttt{5}]\rangle,$$

where 10 is the number of transitions. If we don't care about this number, we write $*$ in its place.

$(IF\ \langle$`(postfix 2 (2 (3 mul add) exec) 1 swap exec sub)`$, [4,5]\rangle)$
$= \langle$`((2 (3 mul add) exec) 1 swap exec sub), [4, 5]`$\rangle$
$\Rightarrow \langle$`(1 swap exec sub), [(2 (3 mul add) exec), 4, 5]`$\rangle$ [*seq*]
$\Rightarrow \langle$`(swap exec sub), [1, (2 (3 mul add) exec), 4, 5]`$\rangle$ [*num*]
$\Rightarrow \langle$`(exec sub), [(2 (3 mul add) exec), 1, 4, 5]`$\rangle$ [*swap*]
$\Rightarrow \langle$`(2 (3 mul add) exec sub), [1, 4, 5]`$\rangle$ [*execute*]
$\Rightarrow \langle$`((3 mul add) exec sub), [2, 1, 4, 5]`$\rangle$ [*num*]
$\Rightarrow \langle$`(exec sub), [(3 mul add), 2, 1, 4, 5]`$\rangle$ [*seq*]
$\Rightarrow \langle$`(3 mul add sub), [2, 1, 4, 5]`$\rangle$ [*execute*]
$\Rightarrow \langle$`(mul add sub), [3, 2, 1, 4, 5]`$\rangle$ [*num*]
$\Rightarrow \langle$`(add sub), [6, 1, 4, 5]`$\rangle$ [*arithop*]
$\Rightarrow \langle$`(sub), [7, 4, 5]`$\rangle$ [*arithop*]
$\Rightarrow \langle$`()`$, [\texttt{-3}, \texttt{5}]\rangle \in FC$ [*arithop*]
$(OF\ \langle$`()`$, [\texttt{-3}, \texttt{5}]\rangle) =$ `-3`

Figure 3.5: An SOS-based execution of a PostFix program.

Not all PostFix executions lead to a final configuration. For example, executing the program `(program 2 add mul 3 4 sub)` on the inputs $[5, 6]$ leads to the configuration $\langle$`(mul 3 4 sub), [11]`$\rangle$. This configuration is not final because there are still commands to be executed. But it does not match the LHS of any rewrite rule consequent. In particular, the [*arithop*] rule requires the stack to have two integers at the top, and here there is only one. This is an example of a stuck state. As discussed earlier, a program reaching a stuck state is considered to signal an error. In this case the error is due to an insufficient number of arguments on the stack.

▷ **Exercise 3.2** Use the SOS for PostFix to determine the values of the PostFix programs in Exercise 1.1. ◁

▷ **Exercise 3.3** Consider extending PostFix with a `rot` command defined by the following rewrite rule:

$$\langle \texttt{rot} \ . \ Q, \ N \ . \ V_0 \ . \ V_1 \ . \ \ldots \ . \ V_N \ . \ S\rangle \Rightarrow \langle Q, \ V_1 \ . \ \ldots \ . \ V_N \ . \ V_0 \ . \ S\rangle, \quad \text{where } (\textit{compare } \texttt{gt } N \ \texttt{0}) \qquad [\textit{rot}]$$

a. Give an informal English description of the behavior of `rot`.

b. What is the contents of the stack after executing the following program on zero arguments?

```
(postfix 0 1 2 3 1 2 3 rot rot rot)
```

c. Using `rot`, write a PostFix executable sequence that serves as subroutine for reversing the top three elements of a given stack.

d. List the kinds of situations in which `rot` can lead to a stuck state, and give a sample program illustrating each one. ◁

▷ **Exercise 3.4** The SOS for PostFix specifies that a configuration is stuck when the stack contains an insufficient number of values for a command. For example, $\langle$`(2 mul)`, $[]\rangle$ is stuck because multiplication requires two stack values.

a. Modify the semantics of PostFix so that, rather than becoming stuck, it uses sensible defaults for the missing values when the stack contains an insufficient number of values. For example, the default value(s) for `mul` would be `1`:

$\langle$`(2 mul)`, $[]\rangle \Rightarrow \langle$`(2)`, $[]\rangle$
$\langle$`(mul)`, $[]\rangle \Rightarrow \langle$`(1)`, $[]\rangle$

b. Do you think this modification is a good idea? Why or why not? ◁

▷ **Exercise 3.5** Suppose the Value domain in the PostFix SOS is augmented with a distinguished error value. Modify the rewrite rules for PostFix so that error configurations push this error value onto the stack. The error value should be "contagious" in the sense that any operation attempting to act on it should also push an error value onto the stack. Under the revised semantics, a program may return a non-error value even though it encounters an error along the way. E.g., `(postfix 0 1 2 add mul 3 4 sub)` should return `-1` rather than signaling an error when called on zero inputs. ◁

▷ **Exercise 3.6** An operational semantics for PostFix2 (the alternative PostFix syntax introduced in Figure 2.9) can be defined by making minor tweaks to the operational semantics for PostFix. Assume that the set of configurations remains unchanged. Then most commands from the secondary syntax can be handled with only cosmetic changes. For example, here is the rewrite rule for a PostFix2 numeral command:

$$\langle \texttt{(int } N\texttt{)} \ . \ Q, \ S\rangle \Rightarrow \langle Q, \ N \ . \ S\rangle \qquad [\textit{numeral}']$$

a. Define an input function that maps PostFix2 programs (which have the form `(postfix2` N C`)`) into an initial configuration.

b. Give rewrite axioms for the PostFix2 commands `(exec)`, `(skip)`, and `(:` C_1 C_2`)`.

(See Exercise 3.7 for another approach to defining the semantics of PostFix2.) ◁

▷ **Exercise 3.7** A distinguishing feature of PostFix2 (the alternative PostFix syntax introduced in Figure 2.9) is that its grammar makes no use of sequence domains. It is reasonable to expect that its operational semantics can be modeled by configurations in which the code component is a single command rather than a command sequence. Based on this idea, design an SOS for PostFix2 in which $CF = \text{Command} \times \text{Stack}$. (Note: do not modify the Command domain.) ◁

▷ **Exercise 3.8** The Hugely Profitable Calculator Company has hired you to design a calculator language called RPN that is based on PostFix. RPN has the same syntax as PostFix command sequences (an RPN program is just a command sequence that is assumed to take zero arguments) and the operations are intended to work in basically the same manner. However, instead of providing a arbitrarily large stack, RPN limits the size of the stack to four values. Additionally, the stack is always **full** in the sense that it contains four values at all times. Initially, the stack contains four `0` values. Pushing a value onto a full stack causes the bottommost stack value to be forgotten. Popping the topmost value from a full stack has the effect of duplicating the bottommost element (i.e., it appears in the last two stack positions after the pop).

a. Develop a complete SOS for the RPN language.

b. Use your SOS to find the results of the following RPN programs:

   i. `(mul 1 add)`

   ii. `(1 20 300 4000 50000 add add add add)`

c. Although PostFix programs are guaranteed to terminate, RPN programs are not. Demonstrate this fact by writing an RPN program that loops infinitely. ◁

▷ **Exercise 3.9** A class of calculators known as *four-function* calculators supports the four usual binary arithmetic operators (`+`, `-`, `*`, `/`) in an infix notation.[7] Here we consider a language FF based on four-function calculators. The programs of FF are any parenthesized sequence of numbers and commands, where commands are `+`, `-`, `*`, `/`, and `=`. The `=` command is used to compute the result of an expression, which may be used as the first argument to another binary operator. The `=` may be elided in a string of operations.

[7] The one described here is based on the TI-1025. See [You81] for more details.

`(1 + 20 =)` $\xrightarrow[FF]{}$ `21`
`(1 + 20 = + 300 =)` $\xrightarrow[FF]{}$ `321`
`(1 + 20 + 300 =)` $\xrightarrow[FF]{}$ `321` {*Note elision of first* `=`.}
`(1 + 20)` $\xrightarrow[FF]{}$`20` {*Last number returned when no final* `=`.}

Other features supported by FF include:

- *Calculation with a constant.* Typing a number followed by `=` uses the number as the first operand in a calculation with the previous operator and second operand:

  `(2 * 5 =)` $\xrightarrow[FF]{}$ `10`
  `(2 * 5 = 7 =)` $\xrightarrow[FF]{}$ `35`
  `(2 * 5 = 7 = 11 =)` $\xrightarrow[FF]{}$ `55`

- *Implied second argument.* If no second argument is specified, the value of the second argument defaults to the first.

  `(5 * =)` $\xrightarrow[FF]{}$ `25`

- *Operator correction.* An operator key can be corrected by typing the correct one after (any number of) unintentional operators.

  `(1 * - + 2 )` $\xrightarrow[FF]{}$ `3`

a. Design an SOS for FF that is consistent with the informal description given above.

b. Use your SOS to find the final values of the following command sequences. (Note: some of the values may be floating point numbers.) Comment on the intended meaning of the unconventional command sequences.

   i. `(8 - 3 + * 4 =)`
   ii. `(3 + 5 / = =)`
   iii. `(3 + 5 / = 6 =)` ◁

3.2.3.3 Progress Rules

Introduction

The commands of PostFix programs are interpreted in a highly linear fashion in Figure 3.4. Even though executable sequences give the code a kind of tree structure, the contents of an executable sequence can only be used when they are prepended to the single stream of commands that is executed by the abstract machine. The fact that the next command to execute is always at the front of this command stream leads to a very simple structure for the rewrite rules in Figure 3.4. Transitions, which appear only in rule consequents, are all of the form

$$\langle C_{first} \ . \ Q, \ S\rangle \Rightarrow \langle Q', \ S'\rangle,$$

where Q' is either the same as Q or is the result of prepending some commands onto the front of Q. In all rules, the command C_{first} at the head of the current command sequence is consumed by the application of the rule.

These simple kinds of rules are not adequate for programming languages exhibiting a more general tree structure. Evaluating a node in an arbitrary syntax tree usually requires the recursive evaluation of its subnodes. For example, consider the evaluation of a sample numerical expression written in the EL language described in Section 2.3:

```
(+ (* (- 5 1) 2) (/ 21 7)).
```

Before the sum can be performed, the results of the product and division must be computed; before the multiplication can be performed, the subtraction must be computed. If the values of operand expressions are computed in a left-to-right order, we expect the evaluation of the expression to occur via the following transition path:

```
(+ (* (- 5 1) 2) (/ 21 7))
⇒ (+ (* 4 2) (/ 21 7))
⇒ (+ 8 (/ 21 7))
⇒ (+ 8 3)
⇒ 11.
```

In each transition, the structure of the expression tree remains unchanged except at the node where the computation is being performed. Rewrite rules for expressing such transitions need to be able to express a transition from tree to tree in terms of transitions between the subtrees. That is, the transition

```
(+ (* (- 5 1) 2) (/ 21 7)) ⇒ (+ (* 4 2) (/ 21 7))
```

is implied by the transition

```
(* (- 5 1) 2) ⇒ (* 4 2),
```

which is in turn is implied by the transition

```
(- 5 1) ⇒ 4.
```

In some sense, "real work" is only done by the last of these transitions; the other transitions just inherit the change because they define the surrounding context in which the change is embedded.

These kinds of transitions on tree-structured programs are expressed by progress rules, which are rules with antecedents. Progress rules effectively allow an evaluation process to reach inside a complicated expression to evaluate one of

its subexpressions. A one-step transition in the subexpression is then reflected as a one-step transition of the expression in which it is embedded.

Example: ELMM

To illustrate progress rules, we will develop an operational semantics for an extremely simple subset of the EL language that we will call ELMM (which stands for EL MINUS MINUS). As shown in Figure 3.6, an ELMM program is just a numerical expression, where a numerical expression is either (1) an integer numeral or (2) an arithmetic operation. There are no arguments, no conditional expressions, and no boolean expressions in ELMM.

Syntactic Domains:

$P \in$ Program
$NE \in$ NumExp
$N \in$ IntegerLiteral $= \{\texttt{-17}, \texttt{0}, \texttt{23}, \ldots\}$
$A \in$ ArithmeticOperator $= \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$

Production Rules:

P	::= `(elmm` NE_{body}`)`	[Program]
NE	::= N_{num}	[IntLit]
	| `(`A_{rator} NE_{rand1} NE_{rand2}`)`	[Arithmetic Operation]

Figure 3.6: An s-expression grammar for ELMM.

In an SOS for ELMM, configurations are just numerical expressions themselves; there are no state components. Numerical literals are the only final configurations. The input and output functions are straightforward. The interesting aspect of the ELMM SOS is the specification of the transition relation $\Rightarrow$, which is shown in Figure 3.7. The ELMM [*arithop*] axiom is similar to the same-named axiom in the POSTFIX SOS; it performs a calculation on integer numerals.

To evaluate expressions with nested subexpressions in a left-to-right order, the rules [*prog-left*] and [*prog-left*] are needed. The [*prog-left*] rule says that if the ELMM abstract machine would make a transition from NE_1 to NE_1', it should also allow a transition from `(arithop` NE_1 NE_2`)` to `(arithop` NE_1' NE_2`)`. This rule permits evaluation of the left operand of the operation while leaving the right operand unchanged. The [*prog-right*] rule is similar, except that it only permits evaluation of the right operand once the left operand has been fully evaluated to an integer numeral. This forces the operands to be evaluated

$$(A\ \ N_1\ \ N_2) \Rightarrow N_{result}, \quad \text{where } N_{result} = (calculate\ A\ N_1\ N_2) \qquad [arithop]$$

$$\frac{NE_1 \Rightarrow NE_1'}{(A\ \ NE_1\ \ NE_2) \Rightarrow (A\ \ NE_1'\ \ NE_2)} \qquad [prog\text{-}left]$$

$$\frac{NE_2 \Rightarrow NE_2'}{(A\ \ N\ \ NE_2) \Rightarrow (A\ \ N\ \ NE_2')} \qquad [prog\text{-}right]$$

Figure 3.7: Rewrite rules defining the transition relation ($\Rightarrow$) for ELMM.

in a left-to-right order. Rules like [*prog-left*] and [*prog-left*] are called **progress rules** because an evaluation step performed on a subexpression allows progress to be made on the evaluation of the whole expression.

In the case of axioms, it was easy to determine the set of transitions that were specified by a rule. But how do we determine exactly what set of transitions are specified by a progress rule? Intuitively, a transition is specified by a progress rule if it matches the consequent of the rule and it's possible to show that the antecedent transition patterns are also satisfied. For example, since the ELMM transition `(- 7 4)` $\Rightarrow$ `3` is justified by the [*arithop*] rule, the transition `(* (- 7 4) (+ 5 6)` $\Rightarrow$ `(* 3 (+ 5 6))` is justified by the [*prog-left*] rule, and the transition `(* 2 (- 7 4))` $\Rightarrow$ `(* 2 3)` is justified by the [*prog-right*] rule. Furthermore, since the above transitions themselves satisfy the antecedents of the [*prog-left*] and [*prog-right*] rules, it is possible to use these rules again to justify the following transitions:

```
(/ (* (- 7 4) (+ 5 6) (% 9 2))) ⇒ (/ (* 3 (+ 5 6) (% 9 2)))
(/ (* 2 (- 7 4)) (% 9 2)) ⇒ (/ (* 2 3) (% 9 2))
(/ 100 (* (- 7 4) (+ 5 6))) ⇒ (/ 100 (* 3 (+ 5 6)))
(/ 100 (* 2 (- 7 4))) ⇒ (/ 100 (* 2 3))
```

These examples suggest that we can justify any transition as long as we can give a proof of the transition based upon the rewrite rules. Such a proof can be visualized as a so-called **proof tree** (also known as a **derivation**) that grows upward from the bottom of the page. The root of a proof tree is the transition we are trying to prove, its intermediate nodes are instantiated progress rules, and its leaves are instantiated axioms. A proof tree is structured so that the consequent of each instantiated rule is one antecedent of its parent (below) in the tree. For example, the proof tree associated with the transition of `(/ 100 (* (- 7 4) (+ 5 6)))` appears in Figure 3.8. We can represent the

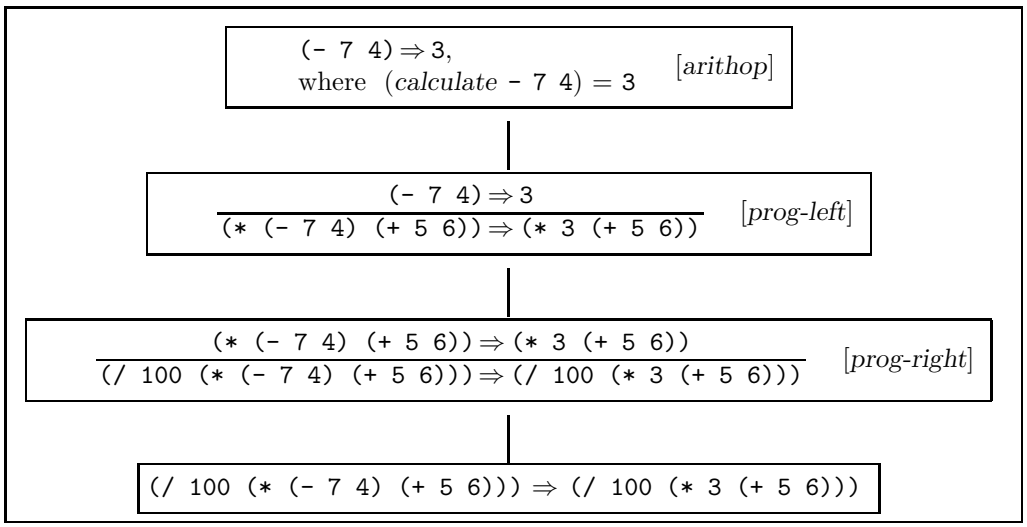


Figure 3.8: A proof tree for a ELMM transition involving nested expressions. The root of the tree is at the bottom of the page; the leaf is at the top.

proof tree in the figure much more concisely by displaying each transition only once, as shown below:

$$\dfrac{\dfrac{\dfrac{}{\texttt{(- 7 4)} \Rightarrow \texttt{3}}\,[\textit{arithop}]}{\texttt{(* (- 7 4) (+ 5 6))} \Rightarrow \texttt{(* 3 (+ 5 6))}}\,[\textit{prog-left}]}{\texttt{(/ 100 (* (- 7 4) (+ 5 6)))} \Rightarrow \texttt{(/ 100 (* 3 (+ 5 6)))}}\,[\textit{prog-right}]$$

The proof tree in this particular example is linear because each of the progress rules involved has only one antecedent transition pattern. A progress rule with n antecedent transition patterns would correspond to a tree node with a branching factor of n. For example, suppose we added the following progress rule to the ELMM SOS:

$$\dfrac{NE_1 \Rightarrow NE_1' \;;\; NE_2 \Rightarrow NE_2'}{\texttt{(arithop } NE_1 \;\; NE_2\texttt{)} \Rightarrow \texttt{(arithop } NE_1' \;\; NE_2'\texttt{)}}\,[\textit{prog-both}]$$

This rule allows simultaneous evaluation of both operands. It leads to proof trees that have branching, such as the following tree in which three arithmetic

operations are performed simultaneously:

$$\dfrac{\dfrac{}{\texttt{(+ 25 75)} \Rightarrow \texttt{100}}\,[arithop] \quad \dfrac{\dfrac{}{\texttt{(- 7 4)} \Rightarrow \texttt{3}}\,[arithop] \quad \dfrac{}{\texttt{(+ 5 6)} \Rightarrow \texttt{11}}\,[arithop]}{\texttt{(* (- 7 4) (+ 5 6))} \Rightarrow \texttt{(* 3 11)}}\,[prog\text{-}both]}{\texttt{(/ (+ 25 75) (* (- 7 4) (+ 5 6)))} \Rightarrow \texttt{(/ 100 (* 3 11))}}\,[prog\text{-}both]$$

It is possible to express any proof tree (even one with branches) in the more traditional linear textual style for a proof. In this style, a proof of a transition is a sequence of transitions where each transition is justified either by an axiom or by a progress rule whose antecedent transitions are justified by transitions earlier in the sequence. A linear textual version of the branching proof tree above would be:

#	**Transition**	**Justification**
[1]	`(+ 25 75)` ⇒`100`	[*arithop*]
[2]	`(- 7 4)` ⇒`3`	[*arithop*]
[3]	`(+ 5 6)` ⇒`11`	[*arithop*]
[4]	`(* (- 7 4) (+ 5 6))` ⇒`(* 3 11)`	[*prog-both*] & [2] & [3]
[5]	`(/ (+ 25 75) (* (- 7 4) (+ 5 6)))` ⇒ `(/ 100 (* 3 11))`	[*prog-both*] & [1] & [4]

The elements of the linear textual proof sequence have been numbered, and justifications involving progress rules include the numbers of the transitions matched by their antecedents. There are many alternative proof sequences for this example that differ in the ordering of the elements. Indeed, the legal linear textual proof sequences for this example are just topological sorts of the original proof tree. Because such linearizations involve making arbitrary choices, we prefer to use the tree based notation, whose structure highlights the essential dependencies in the proof.

When writing down a transition sequence to show the evaluation of an ELMM expresssion we will not explicitly justify every transition with a proof tree, even though such a proof tree must exist. However, if we are listing justifications for transitions, then we will list the names of the rules that would be needed to perform the proof. See Figure 3.9 for an example. (This example uses the original SOS, which does not include the [*prog-both*] rule.)

We shall see in Section 3.4 that the fact that each transition has a proof tree is key to proving properties about transitions. Transition properties are often proven by structural induction on the structure of the proof tree for the transition.

▷ **Exercise 3.10**

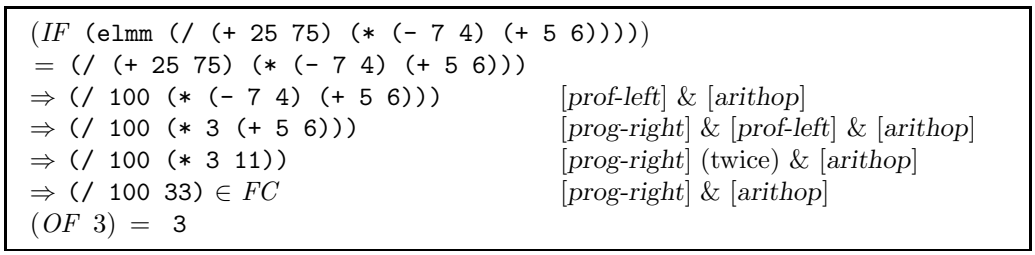


Figure 3.9: An example illustrating evaluation of ELMM expressions.

a. Consider a language ELM (short for EL MINUS) that extends ELMM with indexed references to program inputs. The syntax for ELM is like that of ELMM except that (1) ELM programs have the form `(elm` $N_{numargs}$ NE_{body}`)`, where $N_{numargs}$ specifies the number of expected program arguments and (2) numerical expressions are extended with EL's `(arg` N_{index}`)` construct, which gives the value of the argument whose index is given by N_{index} (assume indices start at 1).

   Write a complete SOS for ELM. Your configurations will need to include a state component representing the program arguments.

b. Write a complete SOS for the full EL language described in Section 2.3.2. You will need to define two kinds of configurations: one to handle numeric expressions and one to handle boolean expressions. Each kind of configuration will be a pair of an expression and a sequence of numeric arguments and will have its own transition relation. ◁

Example: POSTFIX

As another example of progress rules, we will consider an alternative approach for describing the `exec` command of POSTFIX. The [*execute*] axiom in Figure 3.4 handled `exec` by popping an executable sequence off the stack and prepending it to the command sequence following the `exec` command. Figure 3.10 presents a progress rule, [*exec-prog*], that, together with the axiom [*exec-done*], can replace the [*execute*] rule. The [*exec-prog*] rule says that if the abstract machine would make a transition from configuration $\langle Q_{exec},\ S\rangle$ to configuration $\langle Q_{exec}{}',\ S'\rangle$ then it should also allow a transition from the configuration $\langle$`exec .` $Q_{rest},\ Q_{exec}$ `.` $S\rangle$ to $\langle$`exec .` $Q_{rest},\ Q_{exec}{}'$ `.` $S'\rangle$.

Rather than prepending the commands in Q_{exec} to Q_{rest}, the [*exec-prog*] rule effectively executes the commands in Q_{exec} while it remains on the stack. Note that, unlike all the rules that we have seen before, this rule does *not* remove the `exec` command from the current command sequence. Instead, the `exec` command is left in place so that the execution of the command sequence at the

$$\frac{\langle Q_{exec},\ S\rangle \Rightarrow \langle Q_{exec}{}',\ S'\rangle}{\langle \texttt{exec . } Q_{rest},\ Q_{exec}\ \texttt{ . } S\rangle \Rightarrow \langle \texttt{exec . } Q_{rest},\ Q_{exec}{}'\ \texttt{ . } S'\rangle} \quad [\textit{exec-prog}]$$

$$\langle \texttt{exec . } Q_{rest},\ \texttt{() . } S\rangle \Rightarrow \langle Q_{rest},\ S\rangle \quad [\textit{exec-done}]$$

Figure 3.10: A pair of rules that could replace the [*execute*] axiom.

top of the stack will continue during the next transition. Since the commands are removed from Q_{exec} after being executed, the executable sequence at the top of the stack will eventually become empty. At this point, the [*exec-done*] rule takes over, and removes both the completed `exec` command and its associated empty executable sequence.

Figure 3.11 shows how the example considered earlier in Figure 3.1 and Figure 3.5 would be handled using the [*exec-prog*] and [*exec-done*] rules. Each transition is justified by a proof tree that uses the rules listed as a justification. For example, the transition

$$\begin{array}{l}\langle \texttt{(exec sub), [(exec), (mul add), 3, 2, 1, 4, 5]}\rangle \\ \quad \Rightarrow \langle \texttt{(exec sub), [(exec), (add), 6, 1, 4, 5]}\rangle\end{array}$$

is justified by the following proof tree:

$$\dfrac{\dfrac{\dfrac{}{\langle \texttt{(mul add), [3, 2, 1, 4, 5]}\rangle \Rightarrow \langle \texttt{(add), [6, 1, 4, 5]}\rangle}\ [\textit{arithop}]}{\langle \texttt{(exec), [(mul add), 3, 2, 1, 4, 5]}\rangle \Rightarrow \langle \texttt{(exec), [(add), 6, 1, 4, 5]}\rangle}\ [\textit{exec-prog}]}{\begin{array}{c}\langle \texttt{(exec sub), [(exec), (mul add), 3, 2, 1, 4, 5]}\rangle \\ \Rightarrow \langle \texttt{(exec sub), [(exec), (add), 6, 1, 4, 5]}\rangle\end{array}}\ [\textit{exec-prog}]$$

The Meaning of Progress Rules

There are some technical details about progress rules that we glossed over earlier. When we introduced progress rules, we blindly assumed that they were always reasonable. But not all progress rules make sense.

For example, suppose we extend PostFix with a `loop` command defined by the following progress rule:

$$\frac{\langle \texttt{loop . } Q,\ S\rangle \Rightarrow \langle Q,\ S\rangle}{\langle \texttt{loop . } Q,\ S\rangle \Rightarrow \langle Q,\ S\rangle} \quad [\textit{loop}]$$

$(IF\ \langle$`(postfix 2 (2 (3 mul add) exec) 1 swap exec sub)`$, [4,5]\rangle)$
$= \langle$`((2 (3 mul add) exec) 1 swap exec sub), [4, 5]`$\rangle$
$\Rightarrow \langle$`(1 swap exec sub), [(2 (3 mul add) exec), 4, 5]`$\rangle$ [*seq*]
$\Rightarrow \langle$`(swap exec sub), [1, (2 (3 mul add) exec), 4, 5]`$\rangle$ [*num*]
$\Rightarrow \langle$`(exec sub), [(2 (3 mul add) exec), 1, 4, 5]`$\rangle$ [*swap*]
$\Rightarrow \langle$`(exec sub), [((3 mul add) exec), 2, 1, 4, 5]`$\rangle$ [*exec-prog*] & [*num*]
$\Rightarrow \langle$`(exec sub), [(exec), (3 mul add), 2, 1, 4, 5]`$\rangle$ [*exec-prog*] & [*seq*]
$\Rightarrow \langle$`(exec sub), [(exec), (mul add), 3, 2, 1, 4, 5]`$\rangle$ [*exec-prog*] (twice) & [*num*]
$\Rightarrow \langle$`(exec sub), [(exec), (add), 6, 1, 4, 5]`$\rangle$ [*exec-prog*] (twice) & [*arithop*]
$\Rightarrow \langle$`(exec sub), [(exec), (), 7, 4, 5]`$\rangle$ [*exec-prog*] (twice) & [*arithop*]
$\Rightarrow \langle$`(exec sub), [(), 7, 4, 5]`$\rangle$ [*exec-prog*] & [*exec-done*]
$\Rightarrow \langle$`(sub), [7, 4, 5]`$\rangle$ [*exec-done*]
$\Rightarrow \langle$`(), [-3]`$\rangle \in FC$ [*arithop*]
$(OF\ \langle$`(),[-3]`$\rangle) =$ `-3`

Figure 3.11: An example illustrating the alternative rules for `exec`.

Any attempt to prove a transition involving `loop` will fail because there are no axioms involving `loop` with which to terminate the proof tree. Thus, this rule stands for no transitions whatsoever!

We'd like to ensure that all progress rules we consider make sense. We can guarantee this by restricting the form of allowable progress rules to outlaw nonsensical rules like [*loop*]. This so-called **structure restriction** guarantees that any attempt to prove a transition from a given configuration will eventually terminate. The standard structure restriction for an SOS requires the code component of the LHS of each antecedent transition to be a subphrase of the code component of the LHS of the consequent transition. Since program parse trees are necessarily finite, this guarantees that all attempts to prove a transition will have a finite proof.[8]

While simple to follow, the standard structure restriction prohibits many reasonable rules. For example, the [*exec-prog*] rule does not obey this restriction, because the code component of the LHS of the antecedent is unrelated to the code component of the LHS of the consequent. Yet, by considering the entire configuration rather than just the code component, it is possible to design a metric in which the LHS of the antecedent is "smaller" than the LHS of the

[8]This restriction accounts for the term "Structured" in Structured Operational Semantics.

consequent (see Exercise 3.11). While it is sometimes necessary to extend the standard structure restriction in this fashion, most of our rules will actually obey the standard version.

▷ **Exercise 3.11** To guarantee that a progress rule is well-defined, we must show that the antecedent configurations are smaller than the consequent configurations. Here we explore a notion of "smaller than" for the PostFix configurations that establishes the well-definedness of the [*exec-prog*] rule. (Since [*exec-prog*] is the only progress rule for PostFix, it is the only one we need to consider.)

Suppose that we define a relation $<$ on PostFix configurations such that

$$\langle Q_1, \ S\rangle < \langle \texttt{exec} \ . \ Q_2, \ \ Q_1 \ . \ S\rangle$$

for any command sequences Q_1 and Q_2 and any stack S. This is the *only* relation on PostFix configurations; two configurations not satisfying this relation are simply incomparable.

a. A sequence $[a_1, a_2, \ldots]$ is **strictly decreasing** if $a_{i+1} < a_i$ for all i. Using the relation $<$ defined above for configurations, show that every strictly decreasing sequence $[cf_1, cf_2, \ldots]$ of PostFix configurations must be finite.

b. Explain how the result of the previous part implies the well-definedness of the [*exec-prog*] rule. ◁

▷ **Exercise 3.12** The abstract machine for PostFix described thus far employs configurations with two components: a command sequence and a stack. It is possible to construct an alternative abstract machine for PostFix in which configurations consist only of a command sequence. The essence of such a machine is suggested by the transition sequence in Figure 3.12, where the primed rule names are the names of rules for the new abstract machine, not the abstract machine presented earlier.

a. The above example shows that an explicit stack component is not necessary to model PostFix evaluation. Explain how this is possible. (Is there an implicit stack somewhere?)

b. Write an SOS for PostFix in which a configuration is just a command sequence. The SOS should have the behavior exhibited above on the given example. Recall that an SOS has five components; describe all five. Use only axioms to specify your transition relation.

c. In the above example, the `exec` command is handled by replacing it and the executable sequence Q to its left by the contents of Q. This mirrors the prepending behavior of [*execute*] in the original abstract machine. Write rules for the new abstract machine that instead mirror the behavior of [*exec-prog*] and [*exec-done*].

d. Develop an appropriate notion of "smaller than" that establishes the well-definedness of your new [*exec-prog*] rule. (See Exercise 3.11.)

```
((swap exec swap exec) (1 sub) swap (2 mul) swap 3 swap exec)
⇒ ((1 sub) (swap exec swap exec) (2 mul) swap 3 swap exec)  [swap′]
⇒ ((1 sub) (2 mul) (swap exec swap exec) 3 swap exec)       [swap′]
⇒ ((1 sub) (2 mul) 3 (swap exec swap exec) exec)            [swap′]
⇒ ((1 sub) (2 mul) 3 swap exec swap exec)                   [exec′]
⇒ ((1 sub) 3 (2 mul) exec swap exec)                        [swap′]
⇒ ((1 sub) 3 2 mul swap exec)                               [exec′]
⇒ ((1 sub) 6 swap exec)                                     [arithop′]
⇒ (6 (1 sub) exec)                                          [swap′]
⇒ (6 1 sub)                                                 [exec′]
⇒ 5                                                         [arithop′]
```

Figure 3.12: Sample transition sequence for an alternative PostFix abstract machine whose configurations are command sequences.

e. Sketch how you might prove that the new SOS and the original SOS define the behavior. ◁

3.2.3.4 Context-based Semantics

Axioms and progress rules are not the only way to specify the transition relation of a small-step operational semantics. Here we introduce another approach to specifying transitions that is popular in the literature. This approach is based on a notion of **context** that specifies the position of a subphrase in a larger program phrase. Here we will explain this notion and show how it can be used to specify transitions.

In general, a context is a phrase with a single **hole** node in the abstract syntax tree for the phrase. A sample context $\mathbb{C}$ in the ELMM language is `(+ 1 (- □ 2))`, where where □ denotes the hole in the context. "Filling" this hole with any ELMM numerical expression yields another numerical expression. For example, filling $\mathbb{C}$ with `(/ (* 4 5) 3)`, written $\mathbb{C}${`(/ (* 4 5) 3)`}, yields the numerical expression `(+ 1 (- (/ (* 4 5) 3) 2))`.

Contexts are useful for specifying a particular occurrence of a phrase that may occur more than once in an expression. For example, `(+ 3 4)` appears twice in `(* (+ 3 4) (/ (+ 3 4) 2))`. The leftmost occurrence is specified by the context `(* □ (/ (+ 3 4) 2))`, while the rightmost one is specified by `(* (+ 3 4) (/ □ 2))`. Contexts are also useful for specifying the part of a phrase that remains unchanged (the **evaluation context**) when a basic computation (known as a **redex**) is performed. For example, consider the evaluation

of the ELMM expression `(/ 100 (* (- 7 4) (+ 5 6)))`. If operands are evaluated in a left-to-right order, the next redex to be performed is `(- 7 4)`. The evaluation context $\mathbb{E}$ for this redex is `(/ 100 (* □ (+ 5 6)))`. The result of performing the redex (`3` in this case) can be plugged into the evaluation context to yield the result of the transition: $\mathbb{E}\{\texttt{3}\}$ =`(/ 100 (* 3 (+ 5 6)))`.

Evaluation contexts and redexes can be defined via grammars, such as the ones for ELMM in Figure 3.13. In ELMM, a redex is an arithmetic operator applied to two integer numerals. An ELMM evaluation context is either a hole or an arithmetic operation one of whose two operands is an evaluation context. If the evaluation context is in the left operand position ([*Eval Left*]) the right operand can be an arbitrary numerical expression. But if the evaluation context is in the right operand position ([*Eval Right*]), the left operand *must* be a numeral. This structure enforces left-to-right evaluation in ELMM in a way similar to the [*prog-left*] and [*prog-right*] progress rules. Indeed, evaluation contexts are just another way of expressing the information in progress rules — namely, how to find the redex (i.e., where an axiom can be applied).

Redexes

$\mathcal{R} \in$ ElmmRedex

$\mathcal{R}$ ::= `(`$A\ N_1\ N_2$`)` [Arithmetic operation]

Reduction relation ($\rightsquigarrow$)

`(`$A\ N_1\ N_2$`)` $\rightsquigarrow N_{result}$, where N_{result} =$(calculate\ A\ N_1\ N_2)$

Evaluation Contexts

$\mathbb{E} \in$ ElmmEvalContext

$\mathbb{E}$ ::= □ [Hole]
| `(`$A\ \mathbb{E}\ NE$`)` [Eval Left]
| `(`$A\ N\ \mathbb{E}$`)` [Eval Right]

Transition relation ($\Rightarrow$)

$\mathbb{E}\{\mathcal{R}\} \Rightarrow \mathbb{E}\{\mathcal{R}'\}$, where $\mathcal{R} \rightsquigarrow \mathcal{R}'$

Figure 3.13: A context-based specification of the ELMM transition relation.

Associated with redexes is a reduction relation ($\rightsquigarrow$) that corresponds to the basic computations axioms we have seen before. The left hand side of the relation is the redex, while the right hand side is the **reduct**. The transition relation ($\Rightarrow$) is defined in terms of the reduction relation using evaluation contexts: the expression $\mathbb{E}\{\mathcal{R}\}$ rewrites to $\mathbb{E}\{\mathcal{R}'\}$ as long as there is a reduction $\mathcal{R} \rightsquigarrow \mathcal{R}'$. The transition relation is deterministic if there is at most one way to parse an

expression into a evaluation context filled with a redex (which is the case in ELMM).

The following table shows the context-based evaluation of an ELMM expression:

Expression	Evaluation Context	Redex	Reduct
`(/ (+ 25 75) (* (- 7 4) (+ 5 6)))`	`(/ □ (* (- 7 4) (+ 5 6)))`	`(+ 25 75)`	`100`
⇒ `(/ 100 (* (- 7 4) (+ 5 6)))`	`(/ 100 (* □ (+ 5 6)))`	`(- 7 4)`	`3`
⇒ `(/ 100 (* 3 (+ 5 6)))`	`(/ 100 (* 3 □))`	`(+ 5 6)`	`11`
⇒ `(/ 100 (* 3 11))`	`(/ 100 □)`	`(* 3 11)`	`33`
⇒ `(/ 100 33)`	□	`(/ 100 33)`	`3`
⇒ `3`			

Context-based semantics are most convenient in an SOS where the configurations consist solely of a code component. But they can also be adapted to configurations that have state components. For example, Figure 3.14 is a context-based semantics for ELM, the extension to ELMM that includes indexed input via the form `(arg` N_{index}`)` (see Exercise 3.10). An ELM configuration is a pair of (1) an ELM numerical expression and (2) a sequence of numerals representing the program arguments. Both the ELM reduction relation and transition relation must include the program arguments so that the `arg` form can access them.

▷ **Exercise 3.13** Starting wih Figure 3.14, develop a context-based semantics for the full EL language. ◁

▷ **Exercise 3.14** The most natural context-based semantics for PostFix is based on the approach sketched in Exercise 3.12, where configurations consist only of a command sequence. Figure 3.15 is the skeleton of a context-based semantics that defines the transition relation for these configurations. It uses a command sequence context $\mathbb{EQ}$ whose hole can be filled with a command sequence that is internally appended to other command sequences. For example, if $\mathbb{EQ} = [1, 2, \square, \texttt{sub}]$, then $\mathbb{EQ}\{[3, \texttt{swap}]\} = [1, 2, 3, \texttt{swap}, \texttt{sub}]$. Complete the semantics in Figure 3.15 by fleshing out the missing details. ◁

3.3 Big-step Operational Semantics

A small-step operational semantics is a framework for describing program execution as an iterative sequence of small computational steps. But this is not always the most natural way to view execution. We often want to evaluate a phrase

Redexes

$\mathcal{R} \in$ ElmRedex

$\mathcal{R}$ ::= `(`$A\ \ N_1\ \ N_2$`)` [Arithmetic operation]
| `(arg` N_{index}`)` [Indexed Input]

Reduction relation ($\rightsquigarrow$)

$\langle$`(`$A\ \ N_1\ \ N_2$`)`$, N^*\rangle \rightsquigarrow N_{result}$ where $N_{result} =$(*calculate* $A\ \ N_1\ \ N_2$)
$\langle$`(arg` N_{index}`)`$, [N_1, \ldots, N_{N_{size}}]\rangle \rightsquigarrow N_{N_{index}}$
where (*compare* `>` N_{index} `0`) $\wedge \neg$ (*compare* `>` $N_{index}\ \ N_{size}$)

Evaluation Contexts

$\mathbb{E} \in$ ElmEvalContext

$\mathbb{E}$ ::= $\Box$ [Hole]
| `(`$A\ \ \mathbb{E}\ \ NE$`)` [Eval Left]
| `(`$A\ \ N\ \ \mathbb{E}$`)` [Eval Right]

Transition relation ($\Rightarrow$)

$\langle \mathbb{E}\{\mathcal{R}\}, N^*\rangle \Rightarrow \langle \mathbb{E}\{\mathcal{R}'\}, N^*\rangle$ where $\langle \mathcal{R}, N^*\rangle \rightsquigarrow \mathcal{R}'$

Figure 3.14: A context-based specification of the ELM transition relation.

Redexes

$\mathcal{R} \in$ PostFixRedex

$\mathcal{R}$ ::= $[V, \texttt{pop}]$ [Pop]
| $[V_1, V_2, \texttt{swap}]$ [Swap]
| $[N_1, N_2, A]$ [Arithmetic operation]
| ... left as an exercise ...

Reduction relation ($\rightsquigarrow$)

$[V, \texttt{pop}] \rightsquigarrow [\,]$
$[V_1, V_2, \texttt{swap}] \rightsquigarrow [V_2, V_1]$
$[N_1, N_2, A] \rightsquigarrow [N_{result}]$ where $N_{result} =$(*calculate* $A\ \ N_2\ \ N_1$)
... left as an exercise ...

Evaluation Contexts

$\mathbb{EQ} \in$ PostfixEvalSequenceContext

$\mathbb{EQ}$ ::= $V^*\ @\ \Box\ @\ Q$

Transition relation ($\Rightarrow$)

$\mathbb{EQ}\{\mathcal{R}\} \Rightarrow \mathbb{EQ}\{\mathcal{R}'\}$, where $\mathcal{R} \rightsquigarrow \mathcal{R}'$

Figure 3.15: A context-based specification of the transition relation for a subset of PostFix.

by recursively evaluating its subphrases and then combining the results. This is the key idea of denotational semantics, which we shall study in Chapter 4. However, this idea also underlies an alternative form of operational semantics, called **big-step operational semantics (BOS)** (also known as **natural semantics**). Here we briefly introduce big-step semantics in the context of a few examples.

Let's begin by defining a BOS for the simple expression language ELMM, in which programs are numerical expressions that are either numerals or arithmetic operations. A BOS typically has an **evaluation relation** for each non-trivial syntactic domain that directly specifies a result for a given program phrase or configuration. The BOS in Figure 3.16 defines two evaluation relations:

1. $\rightarrow_{NE} \in$ NumExp $\times$ Intlit specifies the evaluation of an ELMM numerical expression; and

2. $\rightarrow_{P} \in$ Program $\times$ Intlit specifies the evaluation of an ELMM program.

$$\frac{NE \rightarrow_{NE} N_{ans}}{\texttt{(elmm } NE\texttt{)} \rightarrow_{P} N_{ans}} \quad [prog]$$

$$N \rightarrow_{NE} N \quad [num]$$

$$\frac{NE_1 \rightarrow_{NE} N_1 \quad ; \quad NE_2 \rightarrow_{NE} N_2}{\texttt{(}A\ NE_1\ NE_2\texttt{)} \rightarrow_{NE} N_{result}} \quad [arithop]$$

where $N_{result} = (calculate\ A\ N_1\ N_2)$

Figure 3.16: Big-step operational semantics for ELMM.

There are two rules specifying $\rightarrow_{NE}$. The [*num*] rule says that numerals evaluate to themselves. The [*arithop*] rule says that evaluating an arithmetic operation $\texttt{(}A\ N_1\ N_2\texttt{)}$ yields the result (N_{result}) of applying the operator to the results (N_1 and N_2) of evaluating the operands. The single [*prog*] rule specifying $\rightarrow_P$ just says that the result of an ELMM program is the result of evaluating its numerical expression.

As with SOS transitions, each instantiation of a BOS evaluation rule is justified by a proof tree, which we shall call an **evaluation tree**. Below is the

proof tree for the evaluation of the program `(elmm (* (- 7 4) (+ 5 6)))`:

$$\dfrac{\dfrac{\dfrac{}{\texttt{7} \longrightarrow_{NE} \texttt{7}}[num] \quad \dfrac{}{\texttt{4} \longrightarrow_{NE} \texttt{4}}[num]}{\texttt{(- 7 4)} \longrightarrow_{NE} \texttt{3}}[arithop] \quad \dfrac{\dfrac{}{\texttt{5} \longrightarrow_{NE} \texttt{5}}[num] \quad \dfrac{}{\texttt{6} \longrightarrow_{NE} \texttt{6}}[num]}{\texttt{(+ 5 6)} \longrightarrow_{NE} \texttt{11}}[arithop]}{\texttt{(* (- 7 4) (+ 5 6))} \longrightarrow_{NE} \texttt{33}}[arithop]}{\texttt{(elmm (* (- 7 4) (+ 5 6)))} \longrightarrow_{P} \texttt{33}}[prog]$$

Unlike the proof tree for an SOS transition, which justifies a single computational step, the proof tree for a BOS transition justifies the entire evaluation! This is the sense in which the steps of a BOS are "big"; they tell how to go from a phrase to an answer (or something close to an answer). In the case of ELMM, the leaves of the proof tree are always trivial evaluations of numerals to themselves.

With BOS evaluations there is no notion of a stuck state. In the ELMM BOS, there is no proof tree for an expression like `(* (/ 7 0) (+ 5 6))` that contains an error. However, we can extend the BOS to include an explicit error token as a possible result and modify the rules to generate and propagate such a token. Since all ELMM programs terminate, a BOS with this extension completely specifies the behavior of a program. But in general, the top-level evaluation rule for a program only partially specifies its behavior, since there is no tree (not even an infinite one) asserting that a program loops. What would the answer A of such a program be in the relation $P \longrightarrow_P A$?

The ELMM BOS rules also do not specify the order in which operands are evaluated, but this is irrelevant anyway since there is no way in ELMM to detect whether one operation is performed before another. The ELMM BOS rules happen to specify a (necessarily deterministic) function, but since they can specify general relations, a BOS can describe non-deterministic evaluation as well.

In ELMM, the evaluation relation maps a code phrase to its result. In general, the LHS (and RHS) of an evaluation relation can be more complex, containing state components in addition to a code component. This is illustrated in the BOS for ELM, which extends ELMM with an indexed input construct (Figure 3.17). Here, the two evaluation relations have different domains than before: they include an integer numeral sequence to model the program arguments.

1. $\longrightarrow_{NE} \in$ (NumExp $\times$ Intlit*) $\times$ Intlit specifies the evaluation of an ELM numerical expression; and

2. $\longrightarrow_{P} \in$ (Program $\times$ Intlit*) $\times$ Intlit specifies the evaluation of an ELM program.

Each of these relations can be read as "evaluating a program phrase relative to the program arguments to yield a result". As a notational convenience, we

$$\frac{NE \xrightarrow{[N_1,\ldots,N_{N_{size}}]}_{NE} N_{ans}}{\texttt{(elm } N_{numargs} \ NE\texttt{)} \xrightarrow{[N_1,\ldots,N_{N_{size}}]}_{P} N_{ans}} \ [prog]$$
where $(compare \texttt{ = } N_{numargs} \ N_{size})$

$$N \xrightarrow{N^*}_{NE} N \quad [num]$$

$$\frac{NE_1 \xrightarrow{N^*}_{NE} N_1 \quad ; \quad NE_2 \xrightarrow{N^*}_{NE} N_2}{\texttt{(}A \ NE_1 \ NE_2\texttt{)} \xrightarrow{N^*}_{NE} N_{result}} \ [arithop]$$
where $N_{result} = (calculate \ A \ N_1 \ N_2)$

$$\texttt{(arg } N_{index}\texttt{)} \xrightarrow{[N_1,\ldots,N_{N_{size}}]}_{NE} N_{N_{index}} \quad [input]$$
where $(compare \texttt{ > } N_{index} \ \texttt{0}) \wedge \neg\, (compare \texttt{ > } N_{index} \ N_{size})$

Figure 3.17: Big-step operational semantics for ELM.

abbreviate $\langle X, N_{args}{}^* \rangle \longrightarrow_X N_{ans}$ as X $\xrightarrow{N_{args}{}^*}_X N_{ans}$, where X ranges over P and NE. The [*prog*] rule is as in ELMM, except that it checks that the number of arguments is as expected and passes them to the body for its evaluation. These arguments are ignored by the [*num*] and [*arithop*] rules, but are used by the [*input*] rule to return the specified argument.

Here is a sample ELM proof tree showing the evaluation of the program `(elm 2 (* (arg 1) (+ 1 (arg 2))))` on the two arguments `7` and `5`:

$$\dfrac{\dfrac{}{\texttt{(arg 1)} \xrightarrow{[7,5]}_{NE} \texttt{7}}\,[input] \quad \dfrac{\dfrac{}{\texttt{1} \xrightarrow{[7,5]}_{NE} \texttt{1}}\,[num] \quad \dfrac{}{\texttt{(arg 2)} \xrightarrow{[7,5]}_{NE} \texttt{5}}\,[input]}{\texttt{(+ (arg 2) 1)} \xrightarrow{[7,5]}_{NE} \texttt{6}}\,[arithop]}{\texttt{(elm 2 (* (arg 1) (+ 1 (arg 2))))} \xrightarrow{[7,5]}_{P} \texttt{42}}\,[prog]$$

Can we describe PostFix execution in terms of a BOS? Yes – via the evaluation relations $\longrightarrow_P$ (for programs) and $\longrightarrow_Q$ (for command sequences) in Figure 3.18. The $\longrightarrow_Q$ relation $\in$ (Commands $\times$ Stack) $\times$ Stack treats command sequences as "stack transformers" that map an input stack to an output stack. We abbreviate $\langle Q, S \rangle \longrightarrow_Q S'$ as $Q \xrightarrow{S}_Q S'$. The [*non-exec*] rule "cheats" by using the SOS transition relation $\Rightarrow$ to specify how a non-`exec` command C transforms the stack to S'. Then $\longrightarrow_Q$ specifies how the rest of the commands transform S' into S''. The [*exec*] rule is more interesting because it uses $\longrightarrow_Q$ in both antecedents. The executable sequence commands Q_{exec} transform S to S', while the remaining commands Q_{rest} transform S' to S''. The [*exec*] rule

illustrates how evaluation order (in this case, executing Q_{exec} before Q_{rest}) can be specified in a BOS by "threading" a state component (in this case, the stack) through an evaluation.

$$\frac{Q \xrightarrow{[N_{N_{size}},\ldots,N_1]}_Q N_{ans} \ . \ S}{\texttt{(postfix } N_{numargs} \ Q\texttt{)} \xrightarrow{[N_1,\ldots,N_{N_{size}}]}_P N_{ans}} \quad [prog]$$
$$\text{where } (compare \texttt{ = } N_{numargs} \ N_{size})$$

$$\frac{\langle C \ . \ Q, \ S\rangle \Rightarrow \langle Q, \ S'\rangle \quad ; \quad Q \xrightarrow{S'}_Q S''}{C \ . \ Q \xrightarrow{S}_Q S''} \quad [non\text{-}exec]$$
$$\text{where } C \neq \texttt{exec}$$

$$\frac{Q_{exec} \xrightarrow{S}_Q S' \quad ; \quad Q_{rest} \xrightarrow{S'}_Q S''}{\texttt{exec} \ . \ Q_{rest} \xrightarrow{(Q_{exec}) \ . \ S}_Q S''} \quad [exec]$$

Figure 3.18: Big-step operational semantics for POSTFIX.

It is convenient to define $\rightarrow_Q$ so that it returns a stack, but stacks are not the final answer we desire. The [*prog*] rule $\in$ (Program $\times$ Intlit*) $\times$ Stack takes care of creating the initial stack from the arguments and extracting the top integer (if it exists) from the final stack.

How do small-step and big-step semantics stack up against each other? Each has its advantages and limitations. A big-step semantics is often more concise than a small-step semantics and one of its proof trees can summarize the entire execution of a program. The recursive nature of a big-step semantics also corresponds more closely to structure of interpreters for high-level languages than a small-step semantics. On the other hand, the iterative step-by-step nature of a small-step semantics corresponds more closely to the way low-level languages are implemented, and it is often a better framework for reasoning about computational resources, errors, and termination. Furthemore, infinite loops are easy to model in a small-step semantics but not in a big-step semantics.

We will use small-step semantics as our default form of operational semantics throughout the rest of this book. This is not because big-step semantics are not useful — they are — but because we will tend to use denotational semantics rather than big-step operational semantics for language specifications that compose the meanings of whole phrases from subphrases.

▷ **Exercise 3.15** Construct a BOS evaluation tree that shows the evaluation of `(postfix 2 (2 (3 mul add) exec) 1 swap exec sub)` on arguments `4` and `5`. ◁

▷ **Exercise 3.16** Extend the BOS in Figure 3.16 to handle the full EL language. You will need a new evaluation relation, $\rightarrow_{BE}$, to handle boolean expressions. ◁

▷ **Exercise 3.17** Modify each of the BOS specifications in Figures 3.16–3.18 to generate and propagate an error token that models signalling an error. Be careful to handle all error situations. ◁

3.4 Operational Reasoning

3.4.1 Programming Language Properties

The suitability of a programming language for a given purpose largely depends on many high-level properties of the language. Important global properties of a programming language include:

- **universality**: the language can express all computable programs;
- **determinism**: the set of possible outcomes from executing a program on any particular inputs is a singleton;
- **termination**: all programs are guaranteed to terminate (i.e., it is not possible to express an infinite loop);
- **static checkability**: a class of program errors can be found by static analysis without resorting to execution;
- **referential transparency**: different occurrences of an expression within the same context always have the same meaning.

Languages often exhibit equivalence properties that allow **safe transformations**: systematic substitutions of one program phrase for another that are guaranteed not to change the behavior of the program. Finally, properties of *particular* programs are often of interest. For instance, we might want to show that a given program terminates, that it uses only bounded resources, or that it is equivalent to some other program. For these sorts of purposes, an important characteristic of a language is how easy it is to prove properties of particular programs written in a language.

A language exhibiting a desired list of properties may not always exist. For example, no language can be both universal and terminating because a universal language must be able to express infinite loops.[9]

[9] But it is often possible to carve a terminating sublanguage out of a universal language.

The properties of a programming language are important to language designers, implementers, and programmers alike. The features included in a language strongly depend on what properties the designers want the language to have. For example, designers of a language in which all programs are intended to terminate cannot include general looping constructs, while designers of a universal language must include features that allow nontermination. Compiler writers extensively use safe transformations to automatically improve the efficiency of programs. The properties of a language influence which language a programmer chooses for a task as well as what style of code the programmer writes.

An important benefit of a formal semantics is that it provides a framework that facilitates proving properties both about the entire language and about particular programs written in the language. Without a formal semantics, our understanding of such properties would be limited to intuitions and informal (and possibly incorrect) arguments. A formal semantics is a shared language for convincing both ourselves and others that some intuition that we have about a program or a language is really true. It can also help us develop new intuitions. It is useful not only to the extent that it helps us construct proofs but also to the extent that it helps us find holes in our arguments. After all, some of the things we think we can prove simply aren't true. The process of constructing a proof can give us important insight into *why* they aren't true.

Below we use operational semantics to reason about EL and PostFix. We first discuss the deterministic behavior of EL under various conditions. Then we show that all PostFix programs are guaranteed to terminate. We conclude by considering conditions under which we can transform one PostFix command sequence to another without changing the behavior of a program.

3.4.2 Deterministic Behavior of EL

Recall that a programming language is deterministic if there is exactly one possible outcome for any pair of program and inputs. In Section 3.2.1, we saw that a deterministic SOS transition relation implies that programs behave deterministically. In Section 3.2.3.1, we argued that the PostFix transition relation is deterministic, so PostFix is a deterministic language.

We can similarly argue that EL is deterministic. We will give the argument for the sublanguage ELMM, but it can be extended to full EL. We will use the SOS for ELMM given in Figure 3.7, which has just three rules: [*arithop*], [*prog-left*], and [*prog-right*]. For a given ELMM numerical expression *NE*, we argue that there is at most one proof tree justifying a transition for *NE*. The proof is by structural induction on the height of the AST for *NE*.

- (Base cases) If *NE* is a numeral, it matches no rules, so there is no transition. If *NE* has the form `(`$A\ \ N_1\ \ N_2$`)`, it can match only the [*arithop*] rule, since there are no transitions involving numerals.
- (Induction cases) *NE* must have the the form `(`$A\ \ NE_1\ \ NE_2$`)`, where at least one of NE_1 and NE_2 is not a numeral. If NE_1 is not a numeral, then *NE* can match only the [*prog-left*] rule, and only in the case where there is a proof tree justifying the transition $NE_1 \Rightarrow NE_1{}'$. By induction, there is at most one such proof tree, so there is at most one proof tree for a transition of *NE*. If NE_1 is a numeral, then NE_2 must not be a numeral, in which case *NE* can match only the [*prog-right*] rule, and similar reasoning applies.

Alternatively, we can prove the determinism of the ELMM transition relation using the context semantics in Figure 3.13. In this case, we need to show that each ELMM numerical expression can be parsed into an evaluation context and redex in at most one way. Such a proof is essentially the same as the one given above, so we omit it.

The ELMM SOS specifies that operations are performed in left-to-right order. Why does the order of evaluation matter? It turns out that it doesn't — there is no way in ELMM to detect the order in which operations are performed! Intuitively, either the evaluation is successful, in which all operations are performed anyway, leading to the same answer, or a divsion/remainder by zero is encountered somewhere along the way, in which case the evaluation is unsuccessful. Note that if we could distinguish between different kinds of errors, the story would be different. For instance, if divide-by-zero gave a different error from remainder-by-zero, then evaluating the expression `(+ (/ 1 0) (% 2 0))` would indicate which of the two subexpressions was evaluated first. The issue of evaluation order is important to implementers, because they sometimes can make program execute more efficiently by reordering operations.

How can we formally show that evaluation order in ELMM does not matter? We begin by replacing the [*prog-right*] rule in the SOS by the following [*prog-right′*] rule to yield a modified ELMM transition relation $\Rightarrow'$.

$$\frac{NE_2 \Rightarrow' NE_2{}'}{(A\ \ NE_1\ \ NE_2) \Rightarrow' (A\ \ NE_1\ \ NE_2{}')} \qquad [\textit{prog-right}']$$

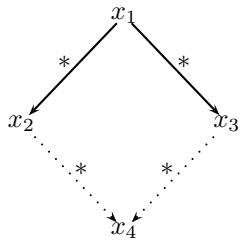
With this change, operands can be evaluated in either order, so the transition relation is no longer deterministic. For example, the expression `(* (- 7 4) (+ 5 6))` now has two transitions:

`(* (- 7 4) (+ 5 6))` $\Rightarrow'$ `(* 3 (+ 5 6))`
`(* (- 7 4) (+ 5 6))` $\Rightarrow'$ `(* (- 7 4) 11)`

Nevertheless, we would like to argue that the behavior of programs is still deterministic even though the transition relation is not.

A handy property for this purpose is called **confluence**. Informally, confluence says that if two transition paths from a configuration diverge, there must be a way to bring them back together. The formal definition is as follows:

> **Confluence:** A relation $\rightarrow \in X \times X$ is confluent if and only if for every $x_1, x_2, x_3 \in X$ such that $x_1 \xrightarrow{*} x_2$ and $x_1 \xrightarrow{*} x_3$, there exists an x_4 such that $x_2 \xrightarrow{*} x_4$ and $x_3 \xrightarrow{*} x_4$. Confluence is usually displayed via the following diagram, in which solid lines are the given relations and the dotted lines are assumed to exist when the property holds. Due to the shape of the diagram, confluence is also called the **diamond property**.



Suppose that a transition relation $\Rightarrow$ is confluent. Then if an initial configuration cf_i has transition paths to two final configurations cf_{f_1} and cf_{f_2}, these are necessarily the same configuration! Why? By confluence, there must be a configuration cf such that $cf_{f_1} \stackrel{*}{\Rightarrow} cf$ and $cf_{f_2} \stackrel{*}{\Rightarrow} cf$. But cf_{f_1} and cf_{f_2} are elements of *Irreducible*, so the only transition paths leaving them have length 0. This means $cf_{f_1} = cf = cf_{f_2}$. Thus, a confluent transition relation guarantees a unique final configuration. Indeed, it guarantees a unique irreducible configuration: it is not possible to get stuck on one path and reach a final configuration on the other.

Confluence by itself does not guarantee a single outcome. It is still possible for a confluent transition relation to have some infinite paths, in which case there is a second outcome (∞). This possibility must be ruled out to prove deterministic behavior. In the case of ELMM, it is easy to prove there are no loops (see Exercise 3.27).

We can now show that ELMM has deterministic behavior under $\Rightarrow'$ by arguing that $\Rightarrow'$ is confluent. We will actually show a stronger property, known as **one-step confluence**, in which the transitive closure stars in the diamond diagram are removed; confluence easily follows from one-step confluence.

Suppose that $NE_1 \Rightarrow' NE_2$ and $NE_1 \Rightarrow' NE_3$. Using terminology from context-based semantics, call the redex reduced in the first transition the "red"

redex and the one reduced in the second transition the "blue" redex. Either these are the same redex, in which case $NE_2 = NE_3$ trivially joins the paths, or the redexes are disjoint (i.e., one does not occur as a subexpression of another). In the latter case, there must be an expression NE_4 that is a copy of NE_1 in which both the red and blue redexes have been reduced. Then $NE_2 \Rightarrow' NE_4$ by reducing the blue redex and $NE_3 \Rightarrow' NE_4$ by reducing the red redex. So NE_4 joins the diverging transitions

We have shown that ELMM has deterministic behavior even when its operations are performed in a non-deterministic order. A similar approach can be used to show that ELM and EL have the same property. Confluence in these languages is fairly straightforward. It becomes much trickier in languages where redexes overlap or performing one redex can copy another.

We emphasize that confluence is a sufficient but not necessary condition for a non-deterministic transition relation to give rise to deterministic behavior. In general, many distinct final configurations might map to the same outcome.

▷ **Exercise 3.18** Suppose that in addition to changing the ELMM SOS by replacing [*prog-right*] with [*prog-right′*], the rule [*prog-both*] introduced on page 58 is added to the SOS.

a. In this modified SOS, how many different transition paths lead from the expression `(/ (+ 25 75) (* (- 7 4) (+ 5 6)))` to the result `3`?

b. Does the modified SOS still have deterministic behavior? Explain your answer ◁

▷ **Exercise 3.19** Consider extending ELMM with a construct `(either` NE_1 NE_2`)` that returns the result of evaluating either NE_1 or NE_2.

a. What are the possible behaviors of the following program?

```
(elmm (* (- (either 1 2) (either 3 4)) (either 5 6)))
```

b. The informal specification of `either` given above is ambiguous. For example, must the expression `(+ (either 1 (/ 2 0)) (either (% 3 0) 4))` return the result `5`, or can it get stuck? The semantics of `either` can be defined either way. Give formal specifications for each interpretation of `either` that is consistent with the informal description. ◁

▷ **Exercise 3.20**

a. Show that the two transition relations (one for NumExp, one for BoolExp) in an EL SOS can be deterministic,

b. Suppose that both transition relations in an EL SOS allow operations to be performed in any order, so that they are non-deterministic. Argue that the behavior of EL programs is still deterministic. ◁

3.4.3 Termination of PostFix Programs

An important property of PostFix is expressed by the following theorem:

> PostFix **Termination Theorem:** All PostFix programs are guaranteed to terminate. That is, executing a PostFix program always either returns a numeral or signals an error.[10]

This theorem is based on the following intuition: existing commands are consumed by execution, but no new commands are ever created, so the commands must eventually "run out." This intuition is essentially correct, but an intuition does not a proof make. After all, PostFix is complex enough to harbor a subtlety that invalidates the intuition. The `nget` command allows the duplication of numerals — is this problematic with regards to termination? Executable sequences are moved to the stack, but their contents can later be prepended to the code component. How can we be certain that this shuffling between code and stack doesn't go on forever? And how do we deal with the fact that executable sequences can be arbitrarily nested?

These questions indicate the need for a more convincing argument that termination is guaranteed. This is the kind of situation in which formal semantics comes in handy. Below we present a proof for termination based on the SOS for PostFix.

3.4.3.1 Energy

Associate with each PostFix configuration a natural number called its *energy* (so called to suggest the potential energy of a dynamical system). By considering each rewrite rule of the semantics in turn, we will prove that the energy strictly decreases with each transition. The energy of an initial configuration must then be an upper bound on the length of any path of transitions leading from the initial configuration. Since the initial energy is finite, there can be no unbounded transition sequences from the initial configuration, so the execution of a program must terminate.

[10] This theorem can fail to hold if PostFix is extended with new commands, such as a `dup` command that duplicates the top stack value. See Section 3.5 for details.

The energy of a configuration is defined by the following energy functions:

$$\begin{aligned}
\mathrm{E}_{config}[\![\langle Q,\ S\rangle]\!] &= \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![S]\!] && (3.1)\\
\mathrm{E}_{seq}[\![[\,]]\!] &= 0 && (3.2)\\
\mathrm{E}_{seq}[\![C\ .\ Q]\!] &= 1 + \mathrm{E}_{com}[\![C]\!] + \mathrm{E}_{seq}[\![Q]\!] && (3.3)\\
\mathrm{E}_{stack}[\![[\,]]\!] &= 0 && (3.4)\\
\mathrm{E}_{stack}[\![V\ .\ S]\!] &= \mathrm{E}_{com}[\![V]\!] + \mathrm{E}_{stack}[\![S]\!] && (3.5)\\
\mathrm{E}_{com}[\![(Q)]\!] &= \mathrm{E}_{seq}[\![Q]\!] && (3.6)\\
\mathrm{E}_{com}[\![C]\!] &= 1, \quad C \text{ not an executable sequence.} && (3.7)
\end{aligned}$$

These definitions embody the following intuitions:

- The energy of a configuration, sequence, or stack is greater than or equal to the sum of the energy of its components.
- Executing a command consumes at least one unit of energy (the 1 that appears in 3.3). This is true even for commands that are transferred from the code component to the stack component (i.e., numerals and executable sequences); such commands are worth one more unit of energy in the command sequence than on the stack.[11]
- An executable sequence can be worth no more energy as a sequence than as a stack value (3.6).

The following lemmas are handy for reasoning about the energy of sequences:

$$\begin{aligned}
\mathrm{E}_{com}[\![C]\!] &\geq 0 && (3.8)\\
\mathrm{E}_{seq}[\![Q_1\ @\ Q_2]\!] &= \mathrm{E}_{seq}[\![Q_1]\!] + \mathrm{E}_{seq}[\![Q_2]\!] && (3.9)
\end{aligned}$$

These can be derived from the energy definitions above. Their derivations are left as an exercise.

Equipped with the energy definitions and identity 3.9, we are ready to prove the PostFix Termination Theorem.

3.4.3.2 The Proof of Termination

Proof: We show that every transition reduces the energy of a configuration. Recall that every transition in an SOS has a proof in terms of the rewrite rules. In the case of PostFix, where all the rules are axioms, the proof is trivial: every

[11] The invocation $\mathrm{E}_{com}[\![V]\!]$ that appears in 3.5 may seem questionable because $\mathrm{E}_{com}[\![]\!]$ should be called on elements of Command, not elements of Value. But since every stack value is also a command, the invocation is well-defined.

PostFix transition is justified by one rewrite axiom. To prove a property about PostFix transitions, we just need to show that it holds for each rewrite axiom in the SOS. Here's the case analysis for the energy reduction property:

- [*num*]: $\langle N\ .\ Q,\ S\rangle \Rightarrow \langle Q,\ N\ .\ S\rangle$

$$\begin{array}{lll} & \mathrm{E}_{config}[\![\langle N\ .\ Q,\ S\rangle]\!] & \\ = & \mathrm{E}_{seq}[\![N\ .\ Q]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.1} \\ = & 1 + \mathrm{E}_{com}[\![N]\!] + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.3} \\ = & 1 + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![N\ .\ S]\!] & \text{by 3.5} \\ = & 1 + \mathrm{E}_{config}[\![\langle Q,\ N\ .\ S\rangle]\!] & \text{by 3.1} \end{array}$$

  The LHS has one more unit of energy than the RHS, so moving a numeral to the stack reduces the configuration energy by one unit.

- [*seq*]: $\langle Q_{exec}\ .\ Q_{rest},\ S\rangle \Rightarrow \langle Q_{rest},\ Q_{exec}\ .\ S\rangle$ Moving an executable sequence to the stack also consumes one energy unit by exactly the same argument as for [*num*].

- [*pop*]: $\langle \texttt{pop}\ .\ Q,\ V_{top}\ .\ S\rangle \Rightarrow \langle Q,\ S\rangle$ Popping V_{top} off of a stack takes at least two energy units:

$$\begin{array}{lll} & \mathrm{E}_{config}[\![\langle \texttt{pop}\ .\ Q,\ V_{top}\ .\ S\rangle]\!] & \\ = & \mathrm{E}_{seq}[\![\texttt{pop}\ .\ Q]\!] + \mathrm{E}_{stack}[\![V_{top}\ .\ S]\!] & \text{by 3.1} \\ = & 1 + \mathrm{E}_{com}[\![\texttt{pop}]\!] + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{com}[\![V_{top}]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.3 and 3.5} \\ = & 2 + \mathrm{E}_{com}[\![V_{top}]\!] + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.7} \\ \geq & 2 + \mathrm{E}_{config}[\![\langle Q,\ S\rangle]\!] & \text{by 3.1 and 3.8} \end{array}$$

- [*swap*]: $\langle \texttt{swap}\ .\ Q,\ V_1\ .\ V_2\ .\ S\rangle \Rightarrow \langle Q,\ V_2\ .\ V_1\ .\ S\rangle$ Swapping the top two elements of a stack consumes two energy units:

$$\begin{array}{lll} & \mathrm{E}_{config}[\![\langle \texttt{swap}\ .\ Q,\ V_1\ .\ V_2\ .\ S\rangle]\!] & \\ = & \mathrm{E}_{seq}[\![\texttt{swap}\ .\ Q]\!] + \mathrm{E}_{stack}[\![V_1\ .\ V_2\ .\ S]\!] & \text{by 3.1} \\ = & 1 + \mathrm{E}_{com}[\![\texttt{swap}]\!] + \mathrm{E}_{seq}[\![Q]\!] & \\ & \quad + \ \mathrm{E}_{com}[\![V_1]\!] + \mathrm{E}_{com}[\![V_2]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.3 and 3.5} \\ = & 2 + \mathrm{E}_{seq}[\![Q]\!] + \mathrm{E}_{stack}[\![V_2\ .\ V_1\ .\ S]\!] & \text{by 3.7 and 3.5} \\ = & 2 + \mathrm{E}_{config}[\![\langle Q,\ V_2\ .\ V_1\ .\ S\rangle]\!] & \text{by 3.1} \end{array}$$

- [*execute*]: $\langle \texttt{exec}\ .\ Q_{rest},\ Q_{exec}\ .\ S\rangle \Rightarrow \langle Q_{exec}\ @\ Q_{rest},\ S\rangle$ Executing the `exec` command consumes two energy units:

$$\begin{array}{lll} & \mathrm{E}_{config}[\![\langle \texttt{exec}\ .\ Q_{rest},\ Q_{exec}\ .\ S\rangle]\!] & \\ = & \mathrm{E}_{seq}[\![\texttt{exec}\ .\ Q_{rest}]\!] + \mathrm{E}_{stack}[\![Q_{exec}\ .\ S]\!] & \text{by 3.1} \\ = & 1 + \mathrm{E}_{com}[\![\texttt{exec}]\!] + \mathrm{E}_{seq}[\![Q_{rest}]\!] & \\ & \quad + \ \mathrm{E}_{com}[\![(Q_{exec})]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.3 and 3.5} \\ = & 2 + \mathrm{E}_{seq}[\![Q_{exec}]\!] + \mathrm{E}_{seq}[\![Q_{rest}]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.6 and 3.7} \\ = & 2 + \mathrm{E}_{seq}[\![Q_{exec}\ @\ Q_{rest}]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.9} \\ = & 2 + \mathrm{E}_{config}[\![\langle Q_{exec}\ @\ Q_{rest},\ S\rangle]\!] & \text{by 3.1} \end{array}$$

- [*nget*], [*arithop*], [*relop-true*], [*relop-false*], [*sel-true*], [*sel-false*]: These cases are similar to those above and are left as exercises for the reader. ◇

The approach of defining a natural number function that decreases on every iteration of a process is a common technique for proving termination. However, inventing the function can sometimes be tricky. In the case of PostFix, we have to get the relative weights of components just right to handle movements between the program and stack.

The termination proof presented above is rather complex. The difficulty is not inherent to PostFix, but is due to the particular way we have chosen to formulate its semantics. There are alternative formulations in which the termination proof is simpler (see exercise 3.25).

▷ **Exercise 3.21** Show that lemmas 3.8 and 3.9 hold. ◁

▷ **Exercise 3.22** Complete the proof of the PostFix termination theorem by showing that the following axioms reduce configuration energy: [*nget*], [*arithop*], [*relop-true*], [*relop-false*], [*sel-true*], [*sel-false*]. ◁

▷ **Exercise 3.23** Bud "eagle-eye" Lojack notices that definitions 3.2 and 3.4 do not appear as the justification for any steps in the PostFix Termination Theorem. He reasons that these definitions are arbitrary, so he could just as well use the following definitions instead:

$$\begin{array}{lcll} \mathrm{E}_{seq}[\![[]]\!] & = & 17 & (\ 3.2'\ ) \\ \mathrm{E}_{stack}[\![[]]\!] & = & 23 & (\ 3.4'\ ) \end{array}$$

Is Bud correct? Explain your answer. ◁

▷ **Exercise 3.24** Prove the termination property of PostFix based on the SOS for PostFix2 from Exercise 3.7.

a. Define an appropriate energy function on configurations in the alternative SOS.

b. Show that each transition in the alternative SOS reduces energy. ◁

3.4.3.3 Structural Induction

The above proof is based on a PostFix SOS that uses only axioms. But what if the SOS contained progress rules, like [*exec-done*] from Section 3.2.3.3? How do we prove a property like reduction in configuration energy when progress rules are involved?

Here's where we can take advantage of the fact that every transition of an SOS must be justified by a finite proof tree based on the rewrite rules. Recall that there are two types of nodes in the proof tree: the leaves, which correspond to axioms, and the intermediate nodes, which correspond to progress rules. Suppose we can show that

- the property holds at each leaf — i.e., it is true for the consequent of every axiom; and
- the property holds at each intermediate node — i.e., for every progress rule, if the property holds for all of the antecedents, then it also holds for the consequent.

Then, by induction on the height of its proof tree, the property must hold for each transition specified by the rewrite rules. This method for proving a property based on the structure of a tree (in this case the proof tree of a transition relation) is called **structural induction**.

As an example of a proof by structural induction, we consider how the previous proof of the termination property for PostFix would be modified for an SOS that uses the [*exec-done*] and [*exec-prog*] rules in place of the [*exec*] rule. It is straightforward to show that the [*exec-done*] axiom reduces configuration energy; this is left as an exercise for the reader. To show that the [*exec-prog*] rule satisfies the property, we must show that *if* its single antecedent transition reduces configuration energy, *then* its consequent transition reduces configuration energy as well.

Recall that the [*exec-prog*] rule has the form:

$$\frac{\langle Q_{exec},\ S\rangle \Rightarrow \langle Q_{exec}{}',\ S'\rangle}{\langle \texttt{exec . } Q_{rest},\ Q_{exec}\ \texttt{ . } S\rangle \Rightarrow \langle \texttt{exec . } Q_{rest},\ Q_{exec}{}'\ \texttt{ . } S'\rangle} \qquad [\textit{exec-prog}]$$

We assume that the antecedent transition,

$$\langle Q_{exec},\ S\rangle \Rightarrow \langle Q_{exec}{}',\ S'\rangle,$$

reduces configuration energy, so that the following inequality holds:

$$\mathrm{E}_{\mathit{config}}[\![\langle Q_{exec},\ S\rangle]\!] > \mathrm{E}_{\mathit{config}}[\![\langle Q_{exec}{}',\ S'\rangle]\!].$$

Then we show that the consequent transition also reduces configuration energy:

$$
\begin{array}{lll}
& \mathrm{E}_{config}[\![\langle \texttt{exec . } Q_{rest},\ Q_{exec}\ \texttt{ . }\ S\rangle]\!] & \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{stack}[\![Q_{exec}\ \texttt{ . }\ S]\!] & \text{by 3.1} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{com}[\![(\,Q_{exec})]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.5} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{seq}[\![Q_{exec}]\!] + \mathrm{E}_{stack}[\![S]\!] & \text{by 3.6} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{config}[\![\langle Q_{exec},\ S\rangle]\!] & \text{by 3.1} \\
> & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{config}[\![\langle Q_{exec}{}',\ S'\rangle]\!] & \text{by assumption} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{seq}[\![Q_{exec}{}']\!] + \mathrm{E}_{stack}[\![S']\!] & \text{by 3.1} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{com}[\![(\,Q_{exec}{}')]\!] + \mathrm{E}_{stack}[\![S']\!] & \text{by 3.6} \\
= & \mathrm{E}_{seq}[\![\texttt{exec . } Q_{rest}]\!] + \mathrm{E}_{stack}[\![Q_{exec}{}'\ \texttt{ . }\ S']\!] & \text{by 3.5} \\
= & \mathrm{E}_{config}[\![\langle \texttt{exec . } Q_{rest},\ Q_{exec}{}'\ \texttt{ . }\ S'\rangle]\!] & \text{by 3.1}
\end{array}
$$

The $>$ appearing in the derivation sequence guarantees that the energy specified by the first line is strictly greater than the energy specified by the last line. This completes the proof that the [*exec-prog*] rule reduces configuration energy. Together with the proofs that the axioms reduce configuration energy, this provides an alternative proof of PostFix's termination property.

▷ **Exercise 3.25** Prove the termination property of PostFix based on the alternative PostFix SOS suggested in Exercise 3.12:

a. Define an appropriate energy function on configurations in the alternative SOS.

b. Show that each transition in the alternative SOS reduces energy.

c. The termination proof for the alternative semantics should be more straightforward than the termination proofs in the text and in Exercise 3.24. What characteristic(s) of the alternative SOS simplify the proof? Does this mean the alternative SOS is a "better" one? ◁

▷ **Exercise 3.26** Prove that the rewrite rules [*exec-prog*] and [*exec-done*] presented in the text specify the same behavior as the [*execute*] rule. That is, show that for any configuration cf of the form $\langle \texttt{exec . } Q,\ S\rangle$, both sets of rules eventually rewrite cf into either (1) a stuck state or (2) the same configuration. ◁

▷ **Exercise 3.27** As in PostFix, every program in the EL language terminates. Prove this fact based on an operational semantics for EL (see Exercise 3.10). ◁

3.4.4 Safe PostFix Transformations

3.4.4.1 Observational Equivalence

One of the most important aspects of reasoning about programs is knowing when it is safe to replace one program phrase by another. Two phrases are said to be

observationally equivalent (or **behaviorally equivalent**) if an instance of one can be replaced by the other in any program without changing the behavior of the program.

Observational equivalence is important because it is the basis for a wide range of program transformation techniques. It is often possible to improve a pragmatic aspect of a program by replacing a phrase by one that is equivalent but more efficient. For example, we expect that the PostFix sequence [`1`, `add`, `2`, `add`] can always be replaced by [`3`, `add`] without changing the meaning of the surrounding program. The latter may be more desirable in practice because it performs fewer additions.

A series of simple transformations can sometimes lead to dramatic performance improvements. Consider the following three transformations on PostFix command sequences, which are just three of the many safe PostFix transformations:

Before	After	Name
$[V_1, V_2, \texttt{swap}]$	$[V_2, V_1]$	[*swap-trans*]
$[(Q), \texttt{exec}]$	Q	[*exec-trans*]
$[N_1, N_2, A]$	$[N_{result}]$ where $N_{result} = (calculate\ A\ N_1\ N_2)$	[*arith-trans*]

Applying these to our running example of a PostFix command sequence yields the following sequence of simplifications:

```
((2 (3 mul add) exec) 1 swap exec sub)
↪ ((2 3 mul add) 1 swap exec sub)        [exec-trans]
↪ ((6 add) 1 swap exec sub)              [arith-trans]
↪ (1 (6 add) exec sub)                   [swap-trans]
↪ (1 6 add sub)                          [exec-trans]
↪ (7 sub)                                [arith-trans]
```

Thus, the original command sequence is a "subtract 7" subroutine. The transformations essentially perform at compile time operations that otherwise would be performed at run time.

It is often tricky to determine whether two phrases are observationally equivalent. For example, at first glance it might seem that the PostFix sequence [`swap`, `swap`] can always be replaced by the empty sequence [ ]. While this transformation is valid in many situations, these two sequences are not observationally equivalent because they behave differently when the stack contains fewer than two elements. For instance, the PostFix program `(postfix 0 1)` returns `1` as a final answer, but the program `(postfix 0 1 swap swap)` generates an error. Two phrases are observationally equivalent only if they are interchangeable in *all* programs.

Observational equivalence can be formalized in terms of the notions of **behavior** and **context** presented earlier. Recall that the behavior of a program

(see Section 3.2.1) is specified by a function *beh* that maps a program and its inputs to a set of possible outcomes:

$$beh : \text{Program} \times \text{Inputs} \rightarrow \mathcal{P}(\text{Outcome})$$

The behavior is deterministic when the resulting set is guaranteed to be a singleton. A program context is a program with a hole in it (see Section 3.2.3.4).

> **Observational Equivalence:** Suppose that $\mathbb{P}$ ranges over program contexts and H ranges over the kinds of phrases that fill the holes in program contexts. Then H_1 and H_2 are defined to be observationally equivalent (written $H_1 =_{obs} H_2$) if and only if for all program contexts $\mathbb{P}$ and all inputs I, $beh\ \langle \mathbb{P}\{H_1\}, I\rangle = beh\ \langle \mathbb{P}\{H_2\}, I\rangle$.

We will consider PostFix as an example. An appropriate notion of program contexts for PostFix is defined in Figure 3.19. A command sequence context $\mathbb{Q}$ is one that can be filled with a sequence of commands to yield another sequence of commands. For example, if $\mathbb{Q} = [\texttt{(2 mul)}, \texttt{3}]\ @\ \Box\ @\ [\texttt{exec}]$, then $\mathbb{Q}\{[\texttt{4}, \texttt{add}, \texttt{swap}]\} = [\texttt{(2 mul)}, \texttt{3}, \texttt{4}, \texttt{add}, \texttt{swap}, \texttt{exec}]$. The [*Prefix*] and [*Suffix*] productions allow the hole to be surrounded by arbitrary command sequences, while the [*Nesting*] production allows the hole to be nested within an executable sequence command. (The notation $[(\mathbb{Q})]$ designates a sequence containing a single element. That element is an executable sequence that contains a single hole.) Due to the presence of @, the grammar for PostfixSequenceContext is ambiguous, but that will not affect our presentation, since filling the hole for any parsing of a sequence context yields exactly the same sequence.

$\mathbb{P} \in$ PostfixProgContext
$\mathbb{Q} \in$ PostfixSequenceContext

$\mathbb{P}$ ::=	`(postfix` $N_{numargs}$ $\mathbb{Q}$`)`	[Program Context]
$\mathbb{Q}$ ::=	$\Box$	[Hole]
	$\mid Q\ @\ \mathbb{Q}$	[Prefix]
	$\mid \mathbb{Q}\ @\ Q$	[Suffix]
	$\mid [(\mathbb{Q})]$	[Nesting]

Figure 3.19: Definition of PostFix contexts.

The possible outcomes of a program must be carefully defined to lead to a satisfactory notion of observational equivalence. The outcomes for PostFix defined in Section 3.2.1 are fine, but small changes can sometimes lead to surprising results. For example, suppose we allow PostFix programs to return the

top value of a non-empty stack, even if the top value is an executable sequence. If we can observe the structure of a returned executable sequence, then this change invalidates all non-trivial program transformations! To see why, take any two sequences we expect to be equivalent (say, [1, add, 2, add] and [3, add]) and plug them into the context (postfix 0 (□)). In the modified semantics, the two outcomes are the executable sequences (1 add 2 add) and (3 add), which are clearly not the same, and so the two sequences are not observationally equivalent.

The problem is that the modified SOS makes distinctions between executable sequence outcomes that are too fine-grained for our purposes. We can fix the problem by instead adopting a coarser-grained notion of behavior in which there is no observable difference between outcomes that are executable sequences. For example, the outcome in this case could be the token executable, indicating that the outcome *is* an executable sequence without divulging *which* particular executable sequence it is. With this change, all the expected program transformations become valid again.

3.4.4.2 Transform Equivalence

It is possible to show the observational equivalence of two particular PostFix command sequences according to the definition on page 84. However, we will follow another route. First, we will develop an easier-to-prove notion of equivalence for PostFix sequences called **transform equivalence**. Then, after giving an example of transform equivalence, we will prove a theorem that transform equivalence implies observational equivalence for PostFix programs. This approach has the advantage that the structural induction proof on contexts needed to show observational equivalence need only be proved once (for the theorem) rather than for every pair of PostFix command sequences.

Transform equivalence is based on the intuition that PostFix command sequences can be viewed as a means of transforming one stack to another. Informally, transform equivalence is defined as follows:

> **Transform Equivalence:** Two PostFix command sequences are **transform equivalent** if they always transform equivalent input stacks to equivalent output stacks.

This definition is informal in that it doesn't say how command sequences can be viewed as transformers or pin down what it means for two stacks to be equivalent. We will now flesh these notions out.

In order to view PostFix command sequences as stack transformers, we will extend the PostFix SOS as follows:

- Modify Stack to contain a distinguished element S_{error}:

  $S \in \text{Stack} = \text{Value}^* + \text{ErrorStack}$
  $\text{ErrorStack} = \{S_{error}\}$

- Extend the transition relation, $\Rightarrow$, so that for all stuck states $cf_{stuck} \in \mathit{Stuck}$, $cf_{stuck} \Rightarrow \langle [\,],\ S_{error} \rangle$. This says that any configuration formerly considered stuck now rewrites to a final configuration with an error stack.

- Define ($\mathit{finalStack}\ Q\ S$) to be S' if $\langle Q,\ S \rangle \stackrel{*}{\Rightarrow} \langle [\,],\ S' \rangle$. The *finalStack* function is well-defined because POSTFIX is deterministic; with the extensions for handling S_{error}, *finalStack* is also a total function.

As examples of *finalStack*, consider ($\mathit{finalStack}$ [`add`, `mul`] [4, 3, 2, 1]) = [24, 1] and ($\mathit{finalStack}$ [`add`, `exec`] [4, 3, 2, 1]) = S_{error}.

The simplest notion of "stack equivalence" is that two stacks are equivalent if they are identical sequences of values. But this notion has problems similar to those discussed above with regard to outcomes in the context of observational equivalence. For example, suppose we are able to show that `(1 add 2 add)` and `(3 add)` are transform equivalent. Then we'd also like the transform equivalence of `((1 add 2 add))` and `((3 add))` to follow as a corollary. But given identical input stacks, these two sequences do *not* yield identical output stacks — the top values of the output stacks are different executable sequences!

To finesse this problem, we need a notion of stack equivalence that treats two executable sequence elements as the same if they are transform equivalent. The recursive nature of these notions prompts us to define *three* mutually recursive equivalence relations that formalize this approach: one between command sequences (transform equivalence), one between stacks (stack equivalence), and one between stack elements (value equivalence).

- Command sequences Q_1 and Q_2 are **transform equivalent** (written $Q_1 \sim_Q Q_2$) if, for all pairs of stack equivalent stacks S_1 and S_2, ($\mathit{finalStack}\ Q_1\ S_1$) is stack equivalent to ($\mathit{finalStack}\ Q_2\ S_2$). The case $S_1 = S_{error} = S_2$ can safely be ignored because S_{error} models only final configurations, not intermediate ones.

- Stacks S_1 and S_2 are **stack equivalent** (written $S_1 \sim_S S_2$) if

  - both S_1 and S_2 are the distinguished error stack, S_{error}; or
  - S_1 and S_2 are equal-length sequences of values that are elementwise value equivalent. I.e., $S_1 = [V_1, \ldots, V_n]$, $S_2 = [V_1{}', \ldots, V_n{}']$, and $V_i \sim_V V_i{}'$ for all i such that $1 \leq i \leq n$.

- Stack elements V_1 and V_2 are **value equivalent** (written $V_1 \sim_V V_2$) if V_1 and V_2 are the same integer numeral (i.e., $V_1 = N = V_2$) or if V_1 and V_2 are executable sequences whose contents are transform equivalent (i.e., $V_1 = (Q_1)$, $V_2 = (Q_2)$, and $Q_1 \sim_Q Q_2$).

Despite the mutually recursive nature of these definitions, we claim that all three are well-defined equivalence relations as long as we choose the largest relations satisfying the descriptions.

Two PostFix command sequences can be proved transform equivalent by case analysis on the structure of input stacks. This is much easier than the case analysis on the structure of contexts that is implied by observational equivalence. Since (as we shall show below) observational equivalence follows from transform equivalence, transform equivalence is a practical technique for demonstrating observational equivalence.

As a simple example of transform equivalence, we show that $[1, \texttt{add}, 2, \texttt{add}] \sim_Q [3, \texttt{add}]$. Consider two non-error stacks S_1 and S_2 such that $S_1 \sim_S S_2$. We proceed by case analysis on the structure of the stacks:

- S_1 and S_2 are both $[\,]$, in which case

$$\begin{array}{ll} & (\mathit{finalStack}\ [3, \texttt{add}]\ [\,]) \\ = & (\mathit{finalStack}\ [\texttt{add}]\ [3]) \\ = & S_{error} \\ = & (\mathit{finalStack}\ [\texttt{add}, 2, \texttt{add}]\ [1]) \\ = & (\mathit{finalStack}\ [1, \texttt{add}, 2, \texttt{add}]\ [\,]) \end{array}$$

- S_1 and S_2 are non-empty sequences whose heads are the same numeric literal and whose tails are stack equivalent. I.e., $S_1 = N\ .\ S_1'$, $S_2 = N\ .\ S_2'$, and $S_1' \sim_S S_2'$.

$$\begin{array}{ll} & (\mathit{finalStack}\ [3, \texttt{add}]\ N\ .\ S_1') \\ = & (\mathit{finalStack}\ [\texttt{add}]\ \texttt{3}\ .\ N\ .\ S_1') \\ = & (\mathit{finalStack}\ [\,]\ N\texttt{+3}\ .\ S_1') \\ = & (\mathit{finalStack}\ [N\texttt{+3}]\ S_1') \\ \sim_S & (\mathit{finalStack}\ [N\texttt{+3}]\ S_2') \\ = & (\mathit{finalStack}\ [\,]\ N\texttt{+3}\ .\ S_2') \\ = & (\mathit{finalStack}\ [\texttt{add}]\ \texttt{2}\ .\ N\texttt{+1}\ .\ S_2') \\ = & (\mathit{finalStack}\ [2, \texttt{add}]\ N\texttt{+1}\ .\ S_2') \\ = & (\mathit{finalStack}\ [\texttt{add}, 2, \texttt{add}]\ \texttt{1}\ .\ N\ .\ S_2') \\ = & (\mathit{finalStack}\ [1, \texttt{add}, 2, \texttt{add}]\ N\ .\ S_2') \end{array}$$

- S_1 and S_2 are non-empty sequences whose heads are transform equivalent executable sequences and whose tails are stack equivalent. I.e., $S_1 = Q_1\ .\ S_1'$, $S_2 = Q_2\ .\ S_2'$, $Q_1 \sim_Q Q_2$, and $S_1' \sim_S S_2'$.

$$
\begin{array}{l}
(\mathit{finalStack}\ [\texttt{3},\texttt{add}]\ \ Q_1\ .\ S_1\,')\\
\quad =\ \ (\mathit{finalStack}\ [\texttt{add}]\ \ \texttt{3}\ .\ Q_1\ .\ S_1\,')\\
\quad =\ \ S_{error}\\
\quad =\ \ (\mathit{finalStack}\ [\texttt{add},\texttt{2},\texttt{add}]\ \ \texttt{1}\ .\ Q_2\ .\ S_2\,')\\
\quad =\ \ (\mathit{finalStack}\ [\texttt{1},\texttt{add},\texttt{2},\texttt{add}]\ \ Q_2\ .\ S_2\,')
\end{array}
$$

In all three cases,

$$(\mathit{finalStack}\ [\texttt{1},\texttt{add},\texttt{2},\texttt{add}]\ \ S_1)\ \sim_S\ (\mathit{finalStack}\ [\texttt{3},\texttt{add}]\ \ S_2)\,,$$

so the transform equivalence of the sequences follows by definition of $\sim_Q$.

We emphasize that stacks can be equivalent without being identical. For instance, given the result of the above example, it is easy to construct two stacks that are stack equivalent without being identical:

$$[\texttt{(1 add 2 add)},5]\ \sim_S\ [\texttt{(3 add)},5].$$

Intuitively, these stacks are equivalent because they cannot be distinguished by any PostFix command sequence. Any such sequence must either ignore both sequence elements (e.g., [pop]), attempt an illegal operation on both sequence elements (e.g., [mul]), or execute both sequence elements on equivalent stacks (via exec). But because the sequence elements are transform equivalent, executing them cannot distinguish them.

3.4.4.3 Transform Equivalence Implies Observational Equivalence

We wrap up the discussion of observational equivalence by showing that transform equivalence of PostFix command sequences implies observational equivalence. This can be explained informally as follows. Every PostFix program context consists of two parts: the commands performed before the hole and the commands performed after the hole. The commands before the hole transform the initial empty stack into S_{pre}. Suppose the hole is filled by one of two executable sequences, Q_1 and Q_2, that are transform equivalent. Then the stacks S_{post1} and S_{post2} that result from executing these sequences, respectively, on S_{pre} must be stack equivalent. The commands performed after the hole must transform S_{post1} and S_{post2} into stack equivalent stacks S_{final1} and S_{final2}. Since behavior depends only on the equivalence class of the final stack, it is impossible to construct a context that distinguishes Q_1 and Q_2. Therefore, they are observationally equivalent.

Below, we present a formal proof that transform equivalence implies observational equivalence.

> PostFix **Transform Equivalence Theorem:** $Q_1 \sim_Q Q_2$ implies $Q_1 =_{obs} Q_2$.

This theorem is useful because it is generally easier to show that two command sequences are transform equivalent than to construct a proof based directly on the definition of observational equivalence.

Proof: We will show that for all sequence contexts $\mathbb{Q}$, $Q_1 \sim_Q Q_2$ implies $\mathbb{Q}\{Q_1\} \sim_Q \mathbb{Q}\{Q_2\}$. The latter equivalence implies that, for all sequence contexts $\mathbb{Q}$ and initial stacks S_{init},

$$(\mathit{finalStack}\ \mathbb{Q}\{Q_1\}\ S_{init}) \sim_S (\mathit{finalStack}\ \mathbb{Q}\{Q_2\}\ S_{init})\,.$$

This in turn implies that for all numerals N_n and arguments sequences $N_{args}{}^*$,

$$\mathit{beh}\ \langle(\texttt{program}\ N_n\ \mathbb{Q}\{Q_1\}), N_{args}{}^*\rangle = \mathit{beh}\ \langle(\texttt{program}\ N_n\ \mathbb{Q}\{Q_2\}), N_{args}{}^*\rangle.$$

So $Q_1 =_{obs} Q_2$ by the definition of observational equivalence.

We will employ the following properties of transform equivalence, which are left as exercises for the reader:

$$Q_1 \sim_Q Q_1{}' \text{ and } Q_2 \sim_Q Q_2{}' \quad \text{implies} \quad Q_1\ @\ Q_2 \sim_Q Q_1{}'\ @\ Q_2{}' \tag{3.10}$$

$$Q_1 \sim_Q Q_2 \quad \text{implies} \quad [(Q_1)] \sim_Q [(Q_2)] \tag{3.11}$$

Property 3.11 is tricky to read; it says that if Q_1 and Q_2 are transform equivalent, then the sequences that result from nesting Q_1 and Q_2 in executable sequences within a singleton sequence are also transform equivalent.

We proceed by structural induction on the grammar of the PostfixSequenceContext domain:

- (Base case) For sequence contexts of the form $\Box$, $Q_1 \sim_Q Q_2$ trivially implies $\Box\{Q_1\} \sim_Q \Box\{Q_2\}$.
- (Induction cases) For each of the following compound sequence contexts, assume that $Q_1 \sim_Q Q_2$ implies $\mathbb{Q}\{Q_1\} \sim_Q \mathbb{Q}\{Q_2\}$ for any $\mathbb{Q}$.
  - For sequence contexts of the form $Q\ @\ \mathbb{Q}$,

    $Q_1 \sim_Q Q_2$

implies	$\mathbb{Q}\{Q_1\} \sim_Q \mathbb{Q}\{Q_2\}$	by assumption
implies	$Q\ @\ (\mathbb{Q}\{Q_1\}) \sim_Q Q\ @\ (\mathbb{Q}\{Q_2\})$	by reflexivity of $\sim_Q$ and 3.10
implies	$(Q\ @\ \mathbb{Q})\{Q_1\} \sim_Q (Q\ @\ \mathbb{Q})\{Q_2\}$	by definition of $\mathbb{Q}$

  - Sequence contexts of the form $\mathbb{Q}\ @\ Q$ are handled similarly to those of the form $Q\ @\ \mathbb{Q}$.
  - For sequence contexts of the form $[(\mathbb{Q})]$,

    $Q_1 \sim_Q Q_2$

implies	$\mathbb{Q}\{Q_1\} \sim_Q \mathbb{Q}\{Q_2\}$	by assumption
implies	$[(\mathbb{Q}\{Q_1\})] \sim_Q [(\mathbb{Q}\{Q_2\})]$	by 3.11
implies	$[(\mathbb{Q})]\{Q_1\} \sim_Q [(\mathbb{Q})]\{Q_2\}$	by definition of $\mathbb{Q}$

    $\diamond$

▷ **Exercise 3.28** For each of the following purported observational equivalences, either prove that the observational equivalence is valid (via transform equivalence), or give a counterexample to show that it is not.

a. $[N, \texttt{pop}] =_{obs} [\,]$

b. $[\texttt{add}, N, \texttt{add}] =_{obs} [N, \texttt{add}, \texttt{add}]$

c. $[N_1, N_2, A] =_{obs} [N_{result}]$, where $N_{result} =$(*calculate* A N_2 N_1)

d. $[\texttt{(}Q\texttt{)}, \texttt{exec}] =_{obs} Q$

e. $[\texttt{(}Q\texttt{)}, \texttt{(}Q\texttt{)}, \texttt{sel}, \texttt{exec}] =_{obs} \texttt{pop . } Q$

f. $[N_1, \texttt{(}N_2\ \texttt{(}Q_a\texttt{)}\ \texttt{(}Q_b\texttt{)}\ \texttt{sel exec)}, \texttt{(}N_2\ \texttt{(}Q_c\texttt{)}\ \texttt{(}Q_d\texttt{)}\ \texttt{sel exec)}, \texttt{sel}, \texttt{exec}]$
$=_{obs} [N_2, \texttt{(}N_1\ \texttt{(}Q_a\texttt{)}\ \texttt{(}Q_c\texttt{)}\ \texttt{sel exec)}, \texttt{(}N_1\ \texttt{(}Q_b\texttt{)}\ \texttt{(}Q_d\texttt{)}\ \texttt{sel exec)}, \texttt{sel}, \texttt{exec}]$

g. $[C_1, C_2, \texttt{swap}] =_{obs} [C_2, C_1]$

h. $[\texttt{swap}, \texttt{swap}, \texttt{swap}] =_{obs} [\texttt{swap}]$ ◁

▷ **Exercise 3.29** Prove lemmas 3.10 and 3.11, which are used to show that transform equivalence implies operational equivalence. ◁

▷ **Exercise 3.30**

a. Modify the PostFix semantics in Figure 3.3 so that the outcome of a PostFix program whose final configuration has an executable sequence at the top is the token `executable`.

b. In your modified semantics, show that transform equivalence still implies observational equivalence. ◁

▷ **Exercise 3.31** Prove the following composition theorem for observationally equivalent PostFix sequences:

$Q_1 =_{obs} Q_1\,'$ and $Q_2 =_{obs} Q_2\,'$ implies Q_1 @ $Q_2 =_{obs} Q_1\,'$ @ $Q_2\,'$ ◁

▷ **Exercise 3.32** Which of the following transformations on EL numerical expressions are safe? Explain your answers. Be sure to consider stuck expressions like `(/ 1 0)`.

a. `(+ 1 2)` $\hookrightarrow$ `3`

b. `(+ 0` *NE*`)` $\hookrightarrow$ *NE*

c. `(* 0` *NE*`)` $\hookrightarrow$ `0`

d. `(+ 1 (+ 2` *NE*`))` $\hookrightarrow$ `(+ 3` *NE*`)`

e. `(+` NE NE`)` $\hookrightarrow$ `(* 2` NE`)`

f. `(if (=` N N`)` NE_1 NE_2`)` $\hookrightarrow$ NE_1

g. `(if (=` NE_1 NE_1`)` NE_2 NE_3`)` $\hookrightarrow$ NE_2

h. `(if` BE NE NE`)` $\hookrightarrow$ NE ◁

▷ **Exercise 3.33†** Develop a notion of transform equivalence for EL that is powerful enough to formally prove that the transformations in Exercise 3.32 that you think are safe are really safe. You will need to design appropriate contexts for EL programs, numerical expressions, and boolean expressions. ◁

▷ **Exercise 3.34‡** Given that transform equivalence implies observational equivalence, it is natural to wonder whether the converse is true. That is, does the following implication hold?

$$Q_1 =_{obs} Q_2 \text{ implies } Q_1 \sim_Q Q_2$$

If so, prove it; if not, explain why. ◁

▷ **Exercise 3.35†** Consider the following $\mathcal{T}_\mathcal{P}$ function, which translates an ELMM program to a PostFix program:

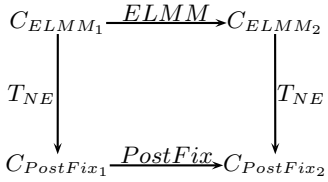
$\mathcal{T}_\mathcal{P}$: $\text{Program}_{ELMM} \rightarrow \text{Program}_{PostFix}$
$\mathcal{T}_\mathcal{P}$ $[\![$`(elmm` NE_{body}`)`$]\!]$ = `(postfix 0` $\mathcal{T}_{\mathcal{NE}}[\![NE_{body}]\!]$ `)`

$\mathcal{T}_{\mathcal{NE}}$: NumExp $\rightarrow$ Commands
$\mathcal{T}_{\mathcal{NE}}[\![N]\!]$ = $[N]$
$\mathcal{T}_{\mathcal{NE}}[\![$`(`$A$ NE_1 NE_2`)`$]\!]$ = $\mathcal{T}_{\mathcal{NE}}[\![NE_1]\!]$ @ $\mathcal{T}_{\mathcal{NE}}[\![NE_2]\!]$ @ $[\mathcal{T}_\mathcal{A}[\![A]\!]$ $]$

$\mathcal{T}_\mathcal{A}$: $\text{ArithmeticOperator}_{ELMM} \rightarrow \text{ArithmeticOperator}_{PostFix}$
$\mathcal{T}_\mathcal{A}[\![$`+`$]\!]$ = `add`
$\mathcal{T}_\mathcal{A}[\![$`-`$]\!]$ = `sub`, *etc.*

a. What is $\mathcal{T}_\mathcal{P}$ $[\![$`(elmm (/ (+ 25 75) (* (- 7 4) (+ 5 6))))`$]\!]$?

b. Intuitively, $\mathcal{T}_\mathcal{P}$ maps an ELMM program to a PostFix program with the same behavior. Develop a proof that formalizes this intuition. As part of your proof, show that the following diagram commutes:



The nodes C_{ELMM_1} and C_{ELMM_2} represent ELMM configurations, and the nodes $C_{PostFix_1}$ and $C_{PostFix_2}$ represent PostFix configurations of the form introduced

in Exercise 3.12. The horizontal arrows are transitions in the respective systems, while the vertical arrows are applications of $\mathcal{T}_{\mathcal{NE}}$. It may help to think in terms of a context-based semantics.

c. Extend the translator to translate (1) ELM programs and (2) EL programs. In each case, prove that the program resulting from your translation has the same behavior as the original program. ◁

3.5 Extending PostFix

We close this chapter on operational semantics by illustrating that slight perturbations to a language can have extensive repercussions for the properties of the language.

You have probably noticed that PostFix has a very limited expressive power. The fact that all programs terminate gives us a hint why. Any language in which all programs terminate can't be universal, because any universal language must allow nonterminating computations to be expressed. Even if we don't care about universality (maybe we just want a good calculator language), PostFix suffers from numerous drawbacks. For example, `nget` allows us to "name" numerals by their position relative to the top of the stack, but these positions change as values are pushed and popped, leading to programs that are challenging to read and write. It would be nicer to give unchanging names to values. Furthermore, `nget` only accesses numerals, and there are situations where we need to access executable sequences and use them more than once.

We could address these problems by allowing executable sequences to be copied from any position on the stack and by introducing a general way to name any value; these extensions are explored in exercises. For now, we will consider extending PostFix with a command that just copies the top value on a stack. Since the top value might be an executable sequence, this at least gives us a way to copy executable sequences — something we could not do before.

Consider a new command, `dup`, which duplicates the value at the top of the stack. After execution of this command, the top two values of the stack will be the same. The rewrite rule for `dup` is given below:

$$\langle \texttt{dup} \ . \ Q, V \ . \ S \rangle \Rightarrow \langle Q, V \ . \ V \ . \ S \rangle \qquad [dup]$$

As a simple example of using `dup`, consider the executable sequence `(dup mul)`, which behaves as a squaring subroutine:

```
(postfix 1 (dup mul) exec) -[12]-> 144
(posttfix 2 (dup mul) dup 3 nget swap exec swap 4 nget swap exec)
  -[5,12]-> 169
```

The introduction of `dup` clearly enhances the expressive power of POSTFIX. But adding this innocent little command has a tremendous consequence for the language: it destroys the termination property! Consider the program `(postfix 0 (dup exec) dup exec)`. Executing this program on zero arguments yields the following transition sequence:

$$\begin{aligned}
&\langle \texttt{((dup exec) dup exec)}, [] \rangle \\
\Rightarrow\ &\langle \texttt{(dup exec)}, [\texttt{(dup exec)}] \rangle \\
\Rightarrow\ &\langle \texttt{(exec)}, [\texttt{(dup exec)}, \texttt{(dup exec)}] \rangle \\
\Rightarrow\ &\langle \texttt{(dup exec)}, [\texttt{(dup exec)}] \rangle \\
\Rightarrow\ &\ldots
\end{aligned}$$

Because the rewrite process returns to a previously visited configuration, it is clear that the execution of this program never terminates.

It is not difficult to see why `dup` invalidates the termination proof from Section 3.4.3. The problem is that `dup` can increase the energy of a configuration in the case where the top element of the stack is an executable sequence. Because `dup` effectively creates new commands in this situation, the number of commands executed can be unbounded.

It turns out that extending POSTFIX with `dup` not only invalidates the termination property, but also results in a language that is universal![12] That is, any computable function can be expressed in POSTFIX+{`dup`}.

This simple example underscores that minor changes to a language can have major consequences. Without careful thought, it is never safe to assume that adding or removing a simple language feature or tweaking a rewrite rule will change a language in only minor ways.

We conclude this chapter with numerous exercises that explore various extensions to the POSTFIX language.

▷ **Exercise 3.36** Extend the POSTFIX SOS so that it handles the following commands:

- `pair`: Let V_1 be the top value on the stack and V_2 be the next to top value. Pop both values off of the stack and push onto the stack a pair object $\langle V_2, V_1 \rangle$.
- `fst`: If the top stack value is a pair $\langle V_{fst}, V_{snd} \rangle$, then replace it with V_{fst}. Otherwise signal an error.
- `right`: If the top stack value is a pair $\langle V_{fst}, V_{snd} \rangle$, then replace it with V_{snd}. Otherwise signal an error. ◁

[12] We are indebted to Carl Witty and Michael Frank for showing us that POSTFIX+{`dup`} is universal.

▷ **Exercise 3.37** Extend the POSTFIX SOS so that it handles the following commands:

- `get`: Call the top stack value v_{index} and the remaining stack values (from top down) v_1, v_2, ..., v_n. Pop v_{index} off the stack. If v_{index} is a numeral i such that $1 \leq i \leq n$, push v_i onto the stack. Signal an error if the stack does not contain at least one value, if v_{index} is not a numeral, or if i is not in the range $[1, n]$. (`get` is like `nget` except that it can copy any value, not just a numeral.)
- `put`: Call the top stack value v_{index}, the next-to-top stack value v_{val}, the remaining stack values (from top down) v_1, v_2, ..., v_n. Pop v_{index} and v_{val} off the stack. If v_{index} is a numeral i such that $1 \leq i \leq n$, change the slot holding v_i on the stack to hold v_{val}. Signal an error if the stack does not contain at least two values, if v_{index} is not a numeral, or if i is not in the range $[1, n]$. ◁

▷ **Exercise 3.38** Write the following programs in POSTFIX+{`dup`}. You may also use the pair commands from Exercise 3.36 and/or the `get`/`put` commands from Exercise 3.37 in your solution, but they are not necessary — for an extra challenge, program purely in POSTFIX+{`dup`}.

a. A program that takes a single argument (call it n) and returns the nth factorial. The factorial f of an integer is a function such that $(f\ 0) = 1$ and $(f\ n) = (n \times (f\ (n-1)))$ for $n \geq 1$.

b. A program that takes a single argument (call it n) and returns the nth Fibonacci number,. The Fibonacci function f is such that $(f\ 0) = 0$, $(f\ 1) = 1$, and $(f\ n) = ((f\ (n-1)) + (f\ (n-2)))$ for $n \geq 2$. ◁

▷ **Exercise 3.39** Abby Stracksen wishes to extend POSTFIX with a simple means of iteration. She suggests that POSTFIX should have a new command of the form `(for` N `(`Q`))`. Abby describes the behavior of her command with the following rewrite axioms:

$$\begin{array}{c} \langle \texttt{(for}\ N\ \texttt{(}Q_{for}\texttt{))}\ .\ Q_{rest},\ S\rangle \\ \Rightarrow \langle N\ .\ Q_{for}\ @\ [\texttt{(for}\ N_{dec}\ \texttt{(}Q_{for}\texttt{))}]\ @\ Q_{rest},\ S\rangle, \\ \text{where}\ \ N_{dec} = (\textit{calculate}\ \texttt{sub}\ N\ \texttt{1}) \\ \text{and}\ \ (\textit{compare}\ \texttt{gt}\ N\ \texttt{0}) \end{array} \qquad [\textit{for-once}]$$

$$\begin{array}{c} \langle \texttt{(for}\ N\ \texttt{(}Q_{for}\texttt{))}\ .\ Q_{rest},\ S\rangle \Rightarrow \langle Q_{rest},\ S\rangle, \\ \text{where}\ \ \neg\,(\textit{compare}\ \texttt{gt}\ N\ \texttt{0}) \end{array} \qquad [\textit{for-done}]$$

Abby calls her extended language POSTLOOP.

a. Give an informal specification of Abby's `for` command that would be appropriate for a reference manual.

b. Using Abby's `for` semantics, what are the results of executing the following POSTLOOP programs when called on zero arguments?

i. `(postloop 0 1 (for 5 (mul)))`

ii. `(postloop 0 1 (for 5 (2 mul)))`

iii. `(postloop 0 1 (for 5 (add mul)))`

iv. `(postloop 0 0 (for 17 (pop 2 add)))`

v. `(postloop 0 0 (for 6 (pop (for 7 (pop 1 add)))))`

c. Extending POSTFIX with the `for` command does not change its termination property. Show this by extending the termination proof described in the notes in the following way:

i. Define the energy of the `for` command.

ii. Show that the transitions in the [*for-once*] and [*for-done*] rules decrease configuration energy.

d. Bud Lojack has developed a `repeat` command of the form `(repeat` N `(`Q`))` that is similar to Abby's `for` command. Bud defines the semantics of his command by the following rewrite rules:

$$\begin{array}{c}\langle \texttt{(repeat } N \texttt{ (}Q_{rpt}\texttt{))} \texttt{ . } Q_{rest},\ S\rangle \\ \Rightarrow \langle N \texttt{ . (repeat } N_{dec} \texttt{ (}Q_{rpt}\texttt{))} \texttt{ . } Q_{rpt} \texttt{ @ } Q_{rest},\ S\rangle, \\ \text{where } N_{dec} = (\textit{calculate } \texttt{sub } N \texttt{ 1}) \\ \text{and } (\textit{compare } \texttt{gt } N \texttt{ 0})\end{array} \qquad [\textit{repeat-once}]$$

$$\begin{array}{c}\langle \texttt{(repeat } N \texttt{ (}Q_{rpt}\texttt{))} \texttt{ . } Q_{rest},\ S\rangle \Rightarrow \langle Q_{rest},\ S\rangle, \\ \text{where } \neg\, (\textit{compare } \texttt{gt } N \texttt{ 0})\end{array} \qquad [\textit{repeat-done}]$$

Does Bud's `repeat` command have the same behavior as Abby's `for` command? That is, does the following observational equivalence hold?

$$[\texttt{(repeat } N \texttt{ (}Q\texttt{))}] =_{obs} [\texttt{(for } N \texttt{ (}Q\texttt{))}]$$

Justify your answer. ◁

▷ **Exercise 3.40** Alyssa P. Hacker has created POSTSAFE, an extension to POSTFIX with a new command called `sdup`: safe `dup`. The `sdup` command is a restricted form of `dup` that does not violoate the termination property of POSTFIX. The informal semantics for `sdup` is as follows: if the top of the stack is a number or a command sequence that doesn't contain `sdup`, duplicate it; otherwise, signal an error.

As a new graduate student in Alyssa's AHRG (Advanced Hacking Research Group), you are assigned to give an operational semantics for `sdup`, and a proof that all POSTSAFE programs terminate. Alyssa set up several intermediate steps to make your life easier.

a. Write the operational semantics rules that describe the behavior of `sdup`. Model the errors through stuck states. You can use the auxiliary function

$$\mathit{contains\_sdup} : \text{Commands} \to \mathit{Bool}$$

that takes a sequence of commands and checks whether it contains `sdup` or not.

b. Consider the product domain $P = \mathbb{N} \times \mathbb{N}$ (recall that $\mathbb{N}$ is the set of natural numbers, starting with 0). On this domain, Alyssa defined the ordering $<_P$ as follows:

Definition 1 (lexicographic order) $\langle a_1, b_1\rangle <_P \langle a_2, b_2\rangle$ *iff*

i. $a_1 < a_2$ *or*

ii. $a_1 = a_2$ *and* $b_1 < b_2$.

E.g., $\langle 3, 10000\rangle <_P \langle 4, 0\rangle$, $\langle 5, 2\rangle <_P \langle 5, 3\rangle$.

Definition 2 *A* strictly decreasing chain *in P is a sequence of elements $p_1, p_2, \ldots$ such that $\forall i\ .\ p_i \in P$ and $\forall i\ .\ p_{i+1} <_P p_i$.*

i. Consider a finite strictly decreasing chain $p_1, p_2, \ldots, p_k$, where $\forall i\ .\ p_i = \langle a_i, b_i\rangle \in P$, such that $k > b_1 + 1$ (i.e., the chain has more than $b_1 + 1$ elements). Prove that $a_k < a_1$.

ii. Show that there is no infinite strictly decreasing chain in P.

c. Prove that each PostSafe program terminates by defining an appropriate energy function $\mathcal{ES}_{\mathit{config}}$. *Note:* If you need to use some helper functions that are intuitively easy to describe but tedious to define (e.g., *contains_sdup*), just give an informal description of them. ◁

▷ **Exercise 3.41** Sam Antix extends the PostFix language to allow programmers to directly manipulate stacks as first-class values. He calls the resulting language StackFix. StackFix adds three commands to the PostFix collection.

- `package`: This command packages a copy of the stack as a first-class value, S. It then clears the stack, leaving S as the only value on the stack.
- `unpackage`: This command pops the top of the stack, which must be a stack-value S, and replaces the stack with an "unpackaged" version of S.
- `switch`: This command pops the top of the stack, which must be a stack-value, S. Then the rest of the stack is packaged (as if by the `package` command); this results in a new stack-value, S_{rest}. Finally, the stack is completely replaced with an "unpackaged" version of S, and the stack-value S_{rest} is pushed on top of the resulting stack. Thus, `switch` effectively switches the roles of the stack-value on top of the stack and the rest of the stack.

As a warm-up, Sam has written some simple StackFix programs. First-class stack values may be returned as the final result of a program execution; in the case, the outcome is the token `stack-value`, which hides the details of the stack value.

`(stackfix 0 1 2 package)` $\xrightarrow{[]}$ `stack-value`
`(stackfix 0 1 2 package unpackage)` $\xrightarrow{[]}$ `2`
`(stackfix 0 1 2 package 3 switch)` $\xrightarrow{[]}$ {*error: top of stack not stack-value*}
`(stackfix 0 1 2 package 3 swap switch)` $\xrightarrow{[]}$ `stack-value`
`(stackfix 0 2 package 3 swap switch pop)` $\xrightarrow{[]}$ `2`
`(stackfix 0 1 2 package 3 swap switch unpackage)` $\xrightarrow{[]}$ `3`

a. Write a definition of the Value domain for the StackFix language.

b. Give transition rules for the `package`, `unpackage`, and `switch` commands.

c. Does `unpackage` add new expressive power to StackFix? If yes, argue why. If no, provide an equivalent sequence of commands from PostFix+{`package`,`switch`}.

d. Does every StackFix program terminate? Give a short, intuitive description of your reasoning. ◁

▷ **Exercise 3.42** Rhea Storr introduces a new PostFix command called `execs` that permits executing a sequence of commands while saving the old stack. She calls her extended language PostSave.

Rhea asks you to help her define transition rules for PostSave that in several steps move $\langle$`execs` . Q, Q_{exec} . $S\rangle$ to the configuration $\langle Q,\ V\ .\ S\rangle$. This sequence of transformations assumes that the configuration $\langle Q_{exec},\ S\rangle$ will eventually result in a final configuration $\langle[]_{\text{Command}},\ V\ .\ S'\rangle$.

Here are some examples that contrast `exec` with `execs`:

`(postsave 0 1 2 (3 mul) exec add)` $\xrightarrow{[]}$ `7`
`(postsave 0 1 2 (3 mul) execs add)` $\xrightarrow{[]}$ `8`
`(postsave 0 (1) execs)` $\xrightarrow{[]}$ `1`
`(postsave 0 2 3 (mul) execs add add)` $\xrightarrow{[]}$ `11`

To implement the SOS for PostSave, Rhea modifies the configuration space:

$cf \in CF = Layer^*$
$L \in Layer = \text{Commands} \times Stack$

Rhea's transition rule for `execs` is:

$$\langle\texttt{execs}\ .\ Q,\ Q_{exec}\ .\ S\rangle\ .\ L^* \Rightarrow \langle Q_{exec},\ S\rangle\ .\ \langle Q,\ S\rangle\ .\ L^* \qquad [execs]$$

Note that the entire stack is copied into the new layer!

a. If $\langle Q,\ S\rangle \xrightarrow{PF} \langle Q',\ S'\rangle$ is a transition rule in PostFix, provide the corresponding rule in PostSave.

b. Provide the rule for an empty command sequence in the top layer.

c. Show that programs in PostSave are no longer guaranteed to terminate by giving a command sequence that is equivalent to `dup`. ◁

▷ **Exercise 3.43** One of the chief limitations of the PostFix language is that there is no way to name values. In this problem, we consider extending PostFix with a simple naming system. We will call the resulting language PostText.

The grammar for PostText is the same as that for PostFix except that there are three new commands:

C ::= ...
| I [Name]
| `def` [Definition]
| `ref` [Name-reference]

Here, I is an element of the syntactic domain Identifier, which includes all alphabetic names except for the PostText command names (`pop`, `exec`, `def`, etc.), which are treated as reserved words of the language.

The model of the PostText language extends the model of PostFix by including a current dictionary as well as a current stack. A dictionary is an object that maintains bindings between names and values. The commands inherited from PostFix have no effect on the dictionary. The informal behavior of the new commands is as follows:

- I: I is a literal name that is similar to an immutable string literal in other languages. Executing this command simply pushes I on the stack. The Value domain must be extended to include identifiers in addition to numerals and executable sequences.
- `def`: Let v_1 be the top stack value and v_2 be the next to top value. The `def` command pops both values off of the stack and updates the current dictionary to include a binding between v_2 and v_1. v_2 should be a name, but v_1 can be any value (including an executable sequence or name literal). It is an error if v_2 is not a name.
- `ref`: The `ref` command pops the top element v_{name} off of the stack, where v_{name} should be a name I. It looks up the value v_{val} associated with I in the current dictionary and pushes v_{val} on top of the stack. It is an error if there is no binding for I in the current dictionary or if v_{name} is not a name.

For example:

```
(posttext 0 average (add 2 div) def 3 7 average ref exec) -[]-> 5
(posttext 0 a 3 def dbl (2 mul) def a ref
   dbl ref exec 4 dbl ref exec add) -[]-> 14
(posttext 0 a b def a ref 7 def b ref) -[]-> 7
(posttext 0 a 5 def a ref 7 def b ref) -[]-> error {5 is not a name.}
(posttext 0 c 4 def d ref 1 add) -[]-> error {d is unbound.}
```

In an SOS for PostText, the usual PostFix configuration space must be extended to include a dictionary object as a new state component:

$$CF_{PostText} \quad = \quad \text{Commands} \times \text{Stack} \times \text{Dictionary}$$

a. Suppose that a dictionary is represented as a sequence of identifier/value pairs:

$$D \in \text{Dictionary} \ = \ (\text{Identifier} \times \text{Value})^*$$

i. Define the final configurations, input function, and output function for the PostText SOS.

ii. Give the rewrite rules for the I, `def`, and `ref` commands.

b. Redo the above problem, assuming that dictionaries are instead represented as functions from identifiers to values, i.e.,

$$D \in \text{Dictionary} \ = \ \text{Identifier} \rightarrow (\text{Value} \ + \ \{\texttt{unbound}\})$$

where `unbound` is a distinguished token indicating an identifier is unbound in the dictionary.

You may find the following *bind* function helpful:

$$\begin{array}{l} \mathit{bind} : \text{Identifier} \rightarrow \text{Value} \rightarrow \text{Dictionary} \rightarrow \text{Dictionary} \\ = \lambda I_{bind}\, VD\, .\ \ \lambda I_{ref}\, .\ \ \textbf{if}\ I_{bind} = I_{ref}\ \ \textbf{then}\ V\ \ \textbf{else}\ (D\ \ I_{ref})\ \ \textbf{fi} \end{array}$$

bind takes a name, a value, and dictionary, and returns a new dictionary in which there is a binding between the name and value in addition to the existing bindings. (If the name was already bound in the given dictionary, the new binding effectively replaces the old.) ◁

▷ **Exercise 3.44** After several focus-group studies, Ben Bitdiddle has decided that PostFix needs a macro facility. Below is Ben's sketch of the informal semantics of the facility for his extended language, which he dubs PostMac.

Macros are specified at the beginning of a PostMac program, as follows:

$$\texttt{(postmac}\ N_{numargs}\ \texttt{((}I_1\ \ V_1\texttt{)}\ \ldots\ \ \texttt{(}I_n\ \ V_n\texttt{))}\ Q\texttt{)}$$

Each macro $\texttt{(}I_i\ \ V_i\texttt{)}$ creates a command, called $I_i \in$ Identifier, that, when executed, pushes the value V_i (which can be an integer or a command sequence) onto the stack. It is illegal to give macros the names of existing PostFix commands, or to use an identifier more than once in a list of macros. The behavior of programs that do so is undefined. Here are some examples Ben has come up with:

`(postmac 0 ((inc (1 add))) (0 inc exec inc exec))` $\xrightarrow{[]}$ `2`

`(postmac 0 ((A 1) (B (2 mul))) (A B exec))` $\xrightarrow{[]}$ `2`

`(postmac 0 ((A 1) (B (2 mul))) (A C exec))` $\xrightarrow{[]}$ `error`
{*undefined macro C*}

`(postmac 0 ((A 1) (B (C mul)) (C 2)) (A B exec))` $\xrightarrow{[]}$ `2`

`(postmac 0 ((A pop)) (1 A))` $\xrightarrow{[]}$ `error`
{*Ill-formed program: macro bodies must be values, not commands*}

Ben started writing an SOS for PostMac, but had to go make a presentation for some venture capitalists. It is your job to complete the SOS.

Before leaving, Ben made the following changes/additions to the domain definitions:

$M \in \text{MacroList} = (\text{Identifier} \times \text{Value})^*$
$P \in \text{Program} = \text{Commands} \times \text{Intlit} \times \text{MacroList}$
$CF = \text{Commands} \times \text{Stack} \times \text{MacroList}$

$C \in \text{Commands} ::= \ldots \mid I$ [Macro Reference]

He also introduced an auxiliary partial function, *lookup*, with the following signature:

$$lookup : \text{Identifier} \times \text{MacroList} \rightharpoonup \text{Value}$$

If *lookup* is given an identifier and a macro list, it returns the value that the identifier is bound to in the macro list. If there is no such value, *lookup* gets stuck.

a. Ben's notes begin the SOS for PostMac as follows:

$$\frac{\langle Q,\ S\rangle \overset{PF}{\Longrightarrow} \langle Q',\ S'\rangle}{\langle Q, S, M\rangle \overset{PM}{\Longrightarrow} \langle Q', S', M\rangle} \qquad \text{[PostFix } \textit{commands}\text{]}$$

where $\overset{PF}{\Longrightarrow}$ is the original transition relation for PostFix and $\overset{PM}{\Longrightarrow}$ is the new transition relation for PostMac. Complete the SOS for PostMac. Your completed SOS should handle the first four of Ben's examples. Don't worry about ill-formed programs. Model errors as stuck states.

b. Louis Reasoner finds out that your SOS handles macros that depend on other macros. He wants to launch a new advertising campaign with the slogan: "Guaranteed to terminate: PostFix with mutually recursive macros!" Show that Louis' new campaign is a bad idea by writing a nonterminating program in PostMac.

c. When Ben returns from his presentation, he finds out you've written a nonterminating program in PostMac. He decides to restrict the language so nonterminating programs are no longer possible. Ben's restriction is that the body (or value) of a macro cannot use any macros. Ben wants you to prove that this restricted language terminates.

   i. Extend the PostFix energy function so that it assigns an energy to configurations that include macros. Fill in the blanks in Ben's definitions of the

functions $\mathrm{E}_{com}[\![C, M]\!]$, $\mathrm{E}_{seq}[\![Q, M]\!]$ and $\mathrm{E}_{stack}[\![S, M]\!]$ and use these functions to define the configuration energy function $\mathrm{E}_{config}[\![\langle Q, S, M\rangle]\!]$.

$$
\begin{aligned}
\mathrm{E}_{com}[\![\texttt{(}Q\texttt{)}, M]\!] &= \mathrm{E}_{seq}[\![Q,\ M]\!] \\
\mathrm{E}_{com}[\![C, M]\!] &= 1 \text{ (C is not an identifier or an executable sequence)} \\
\mathrm{E}_{com}[\![I, M]\!] &= \\
\mathrm{E}_{seq}[\![[\,]_{\text{Command}},\ M]\!] &= 0 \\
\mathrm{E}_{seq}[\![C\ \texttt{.}\ Q,\ M]\!] &= \\
\mathrm{E}_{stack}[\![[\,]_{\text{Value}},\ M]\!] &= 0 \\
\mathrm{E}_{stack}[\![V\ \texttt{.}\ S, M]\!] &= \\
\mathrm{E}_{config}[\![\langle Q, S, M\rangle]\!] &=
\end{aligned}
$$

ii. Use the extended energy function (for the restricted form of POSTMAC) to show that executing a macro decreases the energy of a configuration. Since it is possible to show all the other commands decrease the energy of a configuration (by adapting the termination proof for PostFix without macros), this will show that the restricted form of POSTMAC terminates. ◁

▷ **Exercise 3.45** Dan M. X. Cope, a Lisp hacker, is unsatisfied with POSTTEXT, the name binding extension of POSTFIX introduced in Exercise 3.43. He claims that there is a better way to add name binding to POSTFIX, and creates a brand new language, POSTLISP, to test out his ideas.

The grammar for POSTLISP is the same as that for POSTFIX except that there are four new commands:

C ::= ...
- | I [Name]
- | `bind` [Push new binding]
- | `unbind` [Remove binding]
- | `lookup` [Name lookup]

Here, I is an element of the syntactic domain Identifier, which includes all alphabetic names except for the POSTLISP command names (`pop`, `exec`, `bind`, etc.), which are treated as reserved words of the language.

The model of the POSTLISP language extends the model of POSTFIX by including a *name stack* for each name. A name stack is a stack of values associated with a name that can be manipulated with the `bind`, `unbind`, and `lookup` commands as described below. The commands inherited from POSTFIX have no effect on the name stacks. The informal behavior of the new commands is as follows:

- I: I is a literal name that is similar to an immutable string literal in other languages. Executing this command simply pushes I onto the stack. The Value domain is extended to include names in addition to numerals and executable sequences.

- `bind`: Let v_1 be the top stack value and v_2 be the next-to-top value. The `bind` command pops both values off of the stack and pushes v_1 onto the name stack associated with v_2. Thus v_2 is required to be a name, but v_1 can be any value (including an executable sequence or name literal). It is an error if v_2 is not a name.

- `lookup`: The command `lookup` pops the top element v_{name} off of the stack, where v_{name} should be a name I. If v_{val} is the value at the top of the name stack associated with I, then v_{val} is pushed onto the stack. (v_{val} is *not* popped off of the name stack.) It is an error if the name stack of I is empty, or if v_{name} is not a name.

- `unbind`: The command `unbind` pops the top element v_{name} off of the stack, where v_{name} should be a name I. It then pops the top value off of the name stack associated with I. It is an error if the name stack of I is empty, or if v_{name} is not a name.

- In the initial state, each name is associated with the empty name stack.

For example:

`(postlisp 0 a 3 bind a lookup)` $\xrightarrow{[]}$ `3`
`(postlisp 0 a 8 bind a lookup a lookup add)` $\xrightarrow{[]}$ `16`
`(postlisp 0 a 4 bind a 9 bind a lookup a unbind a lookup add)` $\xrightarrow{[]}$ `13`
`(postlisp 0 19 a bind a lookup)` $\xrightarrow{[]}$ `error` {`19` *is not a name.*}
`(postlisp 0 average (add 2 div) bind 3 7 average lookup exec)` $\xrightarrow{[]}$ `5`
`(postlisp 0 a b bind a lookup 23 bind b lookup)` $\xrightarrow{[]}$ `23`
`(postlisp 0 c 4 bind d lookup 1 add)` $\xrightarrow{[]}$ `error` {`d` *name stack is empty.*}
`(postlisp 0 b unbind)` $\xrightarrow{[]}$ `error` {`b` *name stack is empty*}

In an SOS for PostLisp, the usual PostFix configuration space must be extended to include the name stacks as a new state component. Name stacks are bundled up into an object called a *name file*.

$$CF_{PostLisp} = \text{Commands} \times \text{Stack} \times \text{NameFile}$$
$$F \in \text{NameFile} = \text{Name} \rightarrow \text{Stack}$$

A NameFile is a function mapping a name to the stack of values bound to the name. If F is a name file, then $(F\ I)$ is the stack associated with I in F. The notation $F[I = S]$ denotes a name file that is identical to F except that I is mapped to S.

a. Define the final configurations, input function, and output function for the PostLisp SOS.

b. Give the rewrite rules for the I, `bind`, `unbind`, and `lookup` commands. ◁

▷ **Exercise 3.46** Abby Stracksen is bored with vanilla PostFix (it's not even universal!) and decides to add a new feature, which she calls the *heap*. A heap maps locations to elements from the Value domain, where locations are simply integers:

Location = Intlit

Note that a location can be any integer, including a negative one. Furthermore, integers and locations can be used interchangeably in Abby's language, very much like pointers in pre-ANSI C.

Abby christens her new language PostHeap. The grammar for PostHeap is the same as that for PostFix except that there are three new commands:

C ::= ...
| `allocate` [Allocation]
| `store` [Store in heap location]
| `access` [Access from heap location]

The commands inherited from PostFix have no effect on the heap. The informal behavior of the new commands is as follows:

- `allocate`: Executing this command pushes onto the stack a location that is not used in the heap.
- `store`: Let v_1 be the top stack value and v_2 be the next-to-top value. The `store` command pops v_1 off the stack and writes it into the heap at location v_2. Thus v_1 can be any element from the Value domain and v_2 has to be an Intlit. It is an error if v_2 is not an Intlit. Note that v_2 remains on the stack.
- `access`: Let v_1 be the top stack value. The `access` command reads from the heap at location v_1 and pushes the result onto the stack. Thus v_1 has to be an Intlit. It is an error if v_1 is not an Intlit or if the heap at location v_1 has not been written with `store` before. Note that v_1 remains on the stack.

For example:

`(postheap 0 allocate)` $\xrightarrow{[]}$ *N* {*implementation dependent*}
`(postheap 0 allocate 5 store access)` $\xrightarrow{[]}$ `5`
`(postheap 0 allocate 5 store 4 swap access swap pop add)` $\xrightarrow{[]}$ `9`
`(postheap 0 4 5 store)` $\xrightarrow{[]}$ `4`
`(postheap 0 4 5 store access)` $\xrightarrow{[]}$ `5`
`(postheap 0 access)` $\xrightarrow{[]}$ `error` {*no location given*}
`(postheap 0 allocate access)` $\xrightarrow{[]}$ `error` {*location has not been written*}
`(postheap 0 5 store)` $\xrightarrow{[]}$ `error` {*no location given*}

After sketching this initial description of the heap, Abby realizes that it is already 8:55 on a Friday night and she goes off to watch the X-Files. It is your task to flesh out her initial draft:

a. Give the definition of the Heap domain and the configuration domain *CF*.

b. Let *access-from-heap* be a partial function that, given a Location and a Heap in which Location has been bound, returns an element from the Value domain. In other words, *access-from-heap* has the following signature and definition:

$$\textit{access-from-heap}\text{: Location} \to \text{Heap} \rightharpoonup \text{Value}$$

$(\textit{access-from-heap}\ N\ \langle N, V\rangle\ .\ H) = V$

$(\textit{access-from-heap}\ N_1\ \langle N_2, V\rangle\ .\ H) = (\textit{access-from-heap}\ N_1\ H)$, where $N_1 \neq N_2$

Give the rewrite rules for the `allocate`, `store`, and `access` commands. You may use *access-from-heap*.

c. Is PostHeap a universal programming language? Explain your answer.

d. Abby is concerned about security because PostHeap treats integers and locations interchangeably. Since her programs don't use this "feature", she decides to restrict the language by disallowing pointer arithmetic. She wants to use *tags* to distinguish locations from integers. Abby redefines the Value domain as follows:

$$V \in \text{Value} = (\text{Intlit} \times \text{Tag}) + \text{Command}$$
$$\text{Tag} = \{\texttt{integer}, \texttt{pointer}\}$$

Informally, integers and locations are represented as pairs on the stack: integers are paired with the `integer` tag, while locations are paired with the `pointer` tag.

Give the revised rewrite rules for integers, `add`, `allocate`, `store`, and `access`. ◁

▷ **Exercise 3.47‡** Prove that PostFix+{`dup`} is universal. This can be done by showing how to translate any Turing machine program into a PostFix+{`dup`} program. Assume that integer numerals may be arbitrarily large in magnitude. ◁

Reading

Early approaches to operational semantics defined the semantics of programming languages by translating them to standard abstract machines. Landin's SECD machine [Lan64] is a classic example of such an abstract machine. Plotkin [Plo75] used it to study the semantics of the lambda calculus.

Later, Plotkin introduced Structured Operational Semantics [Plo81] as a more direct approach to specifying an operational semantics. The context-based approach to specifying transition relations for small-step operational semantics was invented by Felleisen and Friedman in [FF86] and explored in a series of papers culminating in [FH92]. Big-step (natural) semantics was introduced by

Kahn in [Kah87]. A concise overview of various approaches to semantics, including several forms of operational semantics, can be found in the first chapter of [Gun92]. The early chapters of [Win93] present an introduction to operational semantics in the context of a simple imperative language.

Other popular forms of operational semantics include **term rewriting systems** ([DJ90, BN98]) and **graph rewriting systems** ([Cou90]).