3.5 IMP++: IMP Extended with Several Features

Our goal here is to challenge the modularity of the basic semantic approaches discussed so far in this chapter, namely big-step SOS, small-step SOS and denotational semantics, by means of a simple programming language design experiment. We shall extend the IMP language in Section 3.1 with several common language features and then attempt to give the resulting language, called IMP++, a formal semantics following each of the approaches. In each case, we aim at reusing the existing semantics of IMP as much as possible. The additional features of IMP++ are the following:

1. A variable increment operation, `++` *Id*, whose role is to infuse side effects in expressions;

2. An input expression construct, `read()`, and an output statement construct, `print(` *AExp* `)`, whose role is to modify the configurations (one needs to add input/output buffers);

3. Abrupt termination, both by means of an explicit `halt` statement and by means of implicit division-by-zero, whose role is to enforce a sudden change of the evaluation context;

4. Spawning a new thread, `spawn` *Stmt*, which executes the given statement concurrently with the rest of the program, sharing all the variables. The role of the `spawn` statement is to test the support of the various semantic approaches for concurrent language features.

5. Blocks allowing local variable declarations, `{` *Stmt* `}` where *Stmt* can include declarations of the form `var` **List{***Id***}** (regarded as ordinary statements). The scope of local declarations is the reminder of the current block. The introduction of blocks with locals begs for changing some of the existing syntax and semantics. For example, there is no need for the current global variable declarations, because they can be replaced by local declarations. The role of this extension is threefold: (1) to demonstrate how execution environment recovery can be achieved in each semantic approach; (2) to generate some non-trivial feature interactions (e.g., spawned threads share the spawning environment); (3) to highlight a language design scenario where the introduction of a new feature may affect the design of the previous ones.

The criterion used for selecting these new features of IMP++ was twofold: on the one hand, these are quite ordinary features encountered in many languages; on the other hand, each of them exposes limitations of one or more of the conventional semantic approaches in this chapter (both before and after this section). Both IMP and IMP++ are admittedly toy languages. However, if a certain programming language semantical style has difficulties in supporting any of the features of IMP or any of the above IMP extensions in IMP++, then one should most likely expect the same problems, but of course amplified, to occur in practical attempts to define real-life programming languages. By "difficulty" we here also mean lack of modularity, that is, that in order to define a new feature one needs to make unrelated changes in the already existing semantics of other features.

IMP++ extends and modifies IMP both syntactically and semantically. Syntactically, it removes from IMP the global declarations and adds the following constructs:

$$
\begin{array}{rcl}
\mathit{AExp} & ::= & \texttt{++}\,\mathit{Id} \quad | \quad \texttt{read()} \\
\mathit{Stmt} & ::= & \texttt{print(}\,\mathit{AExp}\,\texttt{)} \\
 & | & \texttt{halt} \\
 & | & \texttt{spawn}\,\mathit{Stmt} \\
 & | & \{\} \quad | \quad \{\,\mathit{Stmt}\} \quad | \quad \texttt{var}\ \mathbf{List\{}\mathit{Id}\mathbf{\}} \\
\mathit{Pgm} & ::= & \mathit{Stmt}
\end{array}
$$

Semantically, in addition to defining the new language constructs above, we prefer that division-by-zero implicitly halts the program in IMP++, same like the explicit use of `halt`, but in the middle of an expression evaluation. When such an error takes place, one could also generate an error message; however, for simplicity, we do not consider error messages here, only silent termination.

Before we continue with the details of defining each of the new language features in each of the semantics, we mention that there could be various ways to define these. Our goal in this section is to illustrate the lack of modularity of the various semantic styles in different language extension scenarios, and not necessarily to output good error messages. For example, a program that performs a division by zero simply halts its execution when the division by zero takes place.

We first take the various IMP language extensions one by one, discussing what it takes to add each of them to IMP making abstraction of the other features. To make our language design experiment more realistic, when defining each feature we pretend that we do not know what other features will be added. That is, we attempt to achieve local optima for each feature independently. Then, in Section **??**, we put all the features together in the IMP++ language.

3.5.1 Adding Variable Increment

Like in several main-stream programming languages, `++`x increments the value of x in the state and evaluates to the incremented value. This way, the increment operation makes the evaluation of expressions to now have side effects. Consider, for example, the following two programs:

```
var m, n, s ;
n := 100 ;
while (++ m <= n) do (s := s + m)
```

```
var x ;
x := 1 ;
x := ++ x / (++ x / x)
```

The first program shows that the side effect of variable increment can take place anywhere, even in the condition of the while loop; this is actually quite a common programming pattern in languages with variable increment. The second program shows that the addition of side-effects makes the originally intended evaluation strategies of the various expression constructs important. Indeed, recall that for demonstration purposes we originally wanted `+` and `/` to be non-deterministic (i.e., to evaluate their arguments stepwise non-deterministically, possibly interleaving their evaluations), while `<=` to be left-right sequential. These different evaluation strategies can now lead to different program behaviors. For example, the second program above has no fewer than five different behaviors! Indeed, the expression assigned to x can evaluate to 0, 1, 2, 3, and can also perform a division-by-zero. Unfortunately, not all semantic approaches are able to capture all these behaviors.

Big-Step SOS

Big-step SOS is one of the semantics which is the most affected by the inclusion of side effects in expressions, because the previous triples $\langle a,\sigma\rangle \Downarrow \langle i\rangle$ and $\langle b,\sigma\rangle \Downarrow \langle t\rangle$ need to change to four-tuples of the form $\langle a,\sigma\rangle \Downarrow \langle i,\sigma'\rangle$ and $\langle b,\sigma\rangle \Downarrow \langle t,\sigma'\rangle$. These changes are necessary to account for collecting the possible side effects generated by the evaluation of expressions (note that the evaluation of Boolean expressions, because of `<=`, can also have side effects). The big-step SOS of almost all the language constructs needs to change as well. For example, the original big-step SOS of division, namely

$$\frac{\langle a_1,\sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2,\sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ / } a_2,\sigma\rangle \Downarrow \langle i_1 /_{Int}\, i_2\rangle} \quad \text{if } i_2 \neq 0$$

changes as follows:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle \qquad \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 \;\texttt{/}\; a_2, \sigma\rangle \Downarrow \langle i_1 /_{\mathit{Int}} i_2, \sigma_2\rangle}, \quad \text{where } i_2 \neq 0$$

$$\frac{\langle a_1, \sigma_2\rangle \Downarrow \langle i_1, \sigma_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 \;\texttt{/}\; a_2, \sigma\rangle \Downarrow \langle i_1 /_{\mathit{Int}} i_2, \sigma_1\rangle}, \quad \text{where } i_2 \neq 0$$

The rules above make an attempt to capture the intended nondeterministic evaluation strategy of the division operator. We will shortly explain why they fail to fully capture the desired behaviors. One should similarly consider the side effects of expressions in the semantics of statements that need to evaluate expressions. For example, the semantics of the while loop needs to change to propagate the side effects of its condition both when the loop is taken and when the loop is not taken.

Let us next include the big-step semantics of the increment operation; once all the changes to the existing semantics of IMP are applied, the big-step semantics of increment is straightforward:

$$\langle \texttt{++}\, x, \sigma\rangle \Downarrow \langle \sigma(x) +_{\mathit{Int}} 1, \sigma[(\sigma(x) +_{\mathit{Int}} 1)/x]\rangle \qquad \text{(BigStep-Inc)}$$

Indeed, the problem with big-step is not to define the semantics of variable increment, but what it takes to be able to do it. One needs to redefine configurations as explained above and, consequently, to change the semantics of all the already existing features of IMP to use the new configurations. This, and other features defined later on, show how non-modular big-step semantics is.

In addition to being non-modular, big-step SOS cannot properly deal with non-determinism. While it can capture some limited degree of non-determinism as shown above with /, namely it can *non-deterministically choose* which of the subexpressions to evaluate first, it cannot define the full non-deterministic strategy (unless we make radical changes to the definition, such as working with sets of values instead of values, which significantly complicate everything and still fail to capture the non-deterministic behaviors—as it would only capture the non-deterministic evaluation results). To see how the non-deterministic choice evaluation strategy in big-step semantics fails to capture all the desired behaviors, consider the expression $\texttt{++}\,x \,\texttt{/}\, (\texttt{++}\,x \,\texttt{/}\, x)$ with x initially 1, as in the second program at the beginning of Section 3.5.1. This expression can only evaluate to 1, 2 or 3 under non-deterministic choice strategy like we get in big-step SOS. Nevertheless, as explained at the beginning of Section 3.5.1, it could also evaluate to 0 and even perform a division-by-zero under a fully non-deterministic evaluation strategy; we will correctly obtain all these behaviors when using the small-step semantic approaches.

Big-step semantics not only misses behaviors due to its lack of support for non-deterministic evaluation strategies, like shown above, but also hides misbehaviors that it, in principle, detects. For example, assuming $x > 0$, the expression $1 \,\texttt{/}\, (x \,\texttt{/}\, \texttt{++}\,x)$ can either evaluate to 1 or perform an erroneous division by zero. If one searches for all the possible evaluations of a program containing such an expression using the big-step semantics in this section, one will only see the behavior where this expression evaluates to 1; one will never see the erroneous behavior where the division by zero takes place. This will be fixed in Section 3.5.3, where we modify the big-step SOS to support abrupt termination. However, without modifying the semantics, the language designer using big-step semantics may wrongly think that the program is correct. Contrast that with small-step semantics, which, even when one does not add support for abrupt termination, one still detects the wrong behavior by getting stuck on the configuration obtained right before the division by zero.

Additionally, as already explained in Section 3.2.3, the new configurations may be problematic when one wants to execute big-step definitions using rewriting. Indeed, one needs to remove resulting rewrite rules that lead to non-termination, such as rules of the form $R \to R$ corresponding to big-step sequents $R \Downarrow R$ where R are result configurations (e.g., $\langle i, \sigma \rangle$ with $i \in \mathit{Int}$ or $\langle t, \sigma \rangle$ with $t \in \{\texttt{true}, \texttt{false}\}$). We do not use this argument against big-step SOS (its poor modularity is sufficient to disqualify big-step in the competition for an ideal language definitional framework), but rather as a warning to the reader who wants to execute it using rewriting engines (like Maude).

Type System using Big-Step SOS

The typing policy of variable increment is the same as that of variable lookup: provided it has been declared, the incremented variable types to an integer. All we need is to add the following typing rule for increment to the already existing typing rules in Figure 3.10:

$$(xl, x, xl') \vdash \texttt{++}\, x : \mathit{int} \qquad \text{(BigStepTypeSystem-Inc)}$$

Small-Step SOS

Including side effects in expressions is not as bad in small-step semantics as in big-step semantics, because, as discussed in Section 3.3, in small-step SOS one typically uses sequents whose left and right configurations have the same structure even in cases where only some of the configuration components change (e.g., one typically uses sequents of the form $\langle a, \sigma \rangle \to \langle a', \sigma \rangle$ instead of $\langle a, \sigma \rangle \to \langle a' \rangle$); thus, expressions, like any other syntactic categories including statements, can seamlessly modify the state if they need to. However, since we deliberately did not anticipate the inclusion of side effects in expression evaluation, we still have to go back through the existing definition and modify *all* the rules involving expressions to propagate the side effects. For example, the small-step rule

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 \,\texttt{/}\, a_2, \sigma \rangle \to \langle a_1' \,\texttt{/}\, a_2, \sigma \rangle}$$

for the first argument of `/` does not apply anymore when the next step in a_1 is an increment operation (since the state σ changes), so it needs to change to

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma' \rangle}{\langle a_1 \,\texttt{/}\, a_2, \sigma \rangle \to \langle a_1' \,\texttt{/}\, a_2, \sigma' \rangle}$$

Of course, all these changes due to side-effect propagation would have not been necessary if we anticipated that side effects may be added to the language, but the entire point of this exercise is to study the strengths of the various semantic approaches without knowing what comes next.

Once all the changes are applied, one can define the small-step SOS of the increment operation almost identically to its big-step SOS (increment is one atomic step, so small equals big):

$$\langle \texttt{++}\, x, \sigma \rangle \to \langle \sigma(x) +_{\mathit{Int}} 1, \sigma[(\sigma(x) +_{\mathit{Int}} 1)/x] \rangle \qquad \text{(SmallStep-Inc)}$$

Denotational Semantics

The introduction of expression side effects affects denotational semantics even worse than it affects the big-step SOS. Not only that one has to change the denotation of almost every language construct, but the changes are also heavier and more error prone than for big-step SOS. The first change that needs to be made is the type of the denotation functions for expressions:

$$\begin{array}{l}[\![\_]\!] : \mathit{AExp} \to (\mathit{State} \rightharpoonup \mathit{Int})\\ [\![\_]\!] : \mathit{BExp} \to (\mathit{State} \rightharpoonup \mathit{Bool})\end{array}$$

need to change into

$$\begin{array}{l}[\![\_]\!] : \mathit{AExp} \to (\mathit{State} \rightharpoonup \mathit{Int} \times \mathit{State})\\ [\![\_]\!] : \mathit{BExp} \to (\mathit{State} \rightharpoonup \mathit{Bool} \times \mathit{State})\end{array}$$

respectively, to take into account the fact that expressions also return a new possibly changed state besides a result when evaluated. Then one has to change the definitions of the denotation functions for each expression construct to propagate the side effects and to properly extract/combine values and states from/in pairs. For example, the previous denotation function of division, where $[\![a_1 \,/\, a_2]\!]\sigma$ was defined as

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

needs to change to be defined as

$$\begin{cases} (1^{\text{st}}([\![a_1]\!]\sigma) \;/_{\mathit{Int}}\; 1^{\text{st}}([\![a_2]\!](2^{\text{nd}}([\![a_1]\!]\sigma))), 2^{\text{nd}}([\![a_2]\!](2^{\text{nd}}([\![a_1]\!]\sigma))) & \text{if } 1^{\text{st}}([\![a_2]\!](2^{\text{nd}}([\![a_1]\!]\sigma))) \neq 0 \\ \bot & \text{if } 1^{\text{st}}([\![a_2]\!](2^{\text{nd}}([\![a_1]\!]\sigma))) = 0 \end{cases}$$

The above is a bit heavy, repetitive and thus error prone. In implementations of denotational semantics, and sometimes even on paper definitions, one typically uses let binders, or λ-abstractions (see Section 4.5), to bind each subexpression appearing more than once in a denotation function to some variable and then using that variable in each place.

In addition to the denotations of expressions, the denotation functions of all statements except for those of `skip` and sequential composition also need to change, because they involve expressions and need to take their side effects into account. For example, the denotation of the while loop statement `while` b `do` s is the fixed-point of the total function

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

defined as

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \alpha([\![s]\!](2^{\text{nd}}([\![b]\!]\sigma))) & \text{if } 1^{\text{st}}([\![b]\!]\sigma) = \texttt{true} \\ 2^{\text{nd}}([\![b]\!]\sigma) & \text{if } 1^{\text{st}}([\![b]\!]\sigma) = \texttt{false} \\ \bot & \text{if } [\![b]\!]\sigma = \bot \end{cases}$$

All the changes above were necessary to support the side effects generated by the increment construct. The denotational semantics of programs does not need to change. All programs that make no use of increment should still have exactly the same semantics as before (to be precise, the program denotation functions are different as syntactic terms, but they evaluate to the same values when invoked). We are now ready to give the denotational semantics of increment:

$$[\![\texttt{++}\,x]\!]\sigma = \begin{cases} (\sigma(x) +_{\mathit{Int}} 1, \sigma[\sigma(x) +_{\mathit{Int}} 1/x]) & \text{if } \sigma(x) \neq \bot \\ \bot & \text{if } \sigma(x) = \bot \end{cases} \qquad \text{(DENOTATIONAL-INC)}$$

Like for big-step SOS, giving the denotational semantics of increment is not difficult; the difficulty stays in what it takes to be able to do so.

Needless to say that denotational semantics, as we used it here, is very non-modular. The brute force approach above is the most straightforward approach when one's goal is to exclusively add increment to IMP—recall that our experiment in this section assumes that each language extension is the last one. When one expects many extensions to a language that in an operational setting would yield changes to the program configuration, in denotational semantics one is better served using a *continuation* based or a *monadic* style. These styles were briefly mentioned in Section 3.4.3 and further discussed in Section 3.10. They are more involved, and thus less accessible to non-expert language designers. Moreover, switching to such styles is a radical change to a language definition, which requires a complete redefinition of the language. It is therefore highly recommended that one starts directly with a continuation or monadic style if one expects many and non-trivial language extensions. We here, however, prefer the straightforward denotational approach because it is easier to understand and because our overall focus of this book is more operational than denotational.

Besides lacking modularity, the denotational semantics above also lacks non-determinism. Indeed, note that, for example, our denotation of division first evaluates the first expression and then the second expression. Since expressions have side effects, different orders of evaluation can lead to different behaviors. Since the denotations of expressions are partial functions, they cannot have two different behaviors in the same state; therefore, unlike in big-step SOS, we cannot simply add another equation for the other order of evaluation because that would yield an inconsistent equational theory. The consecrated method to define non-determinism in denotational semantics is to use *powerdomains*, as briefly discussed in Section 3.4.3 and further discussed in Section 3.10. Like using continuations or monads, the use of powerdomains also requires a complete redesign of the entire semantics and makes it less accessible to non-experts. Moreover, one cannot obtain a feasible executable model of the language anymore, because the use of powerdomains requires to collect all possible behaviors of any fragment of program at any point in execution; this will significantly slow down the execution of the semantics, making it, for example, infeasible or even unusable as an interpreter anymore.

3.5.2 Adding Input/Output

The semantics of the input expression construct `read()` is that it consumes the next integer from the "input buffer" and evaluates to that integer. The semantics of the output statement construct `print(`a`)` is that a is first evaluated to some integer, which is then collected in an "output buffer". By a "buffer" we here mean some list structure over integers. The semantics of the input/output constructs will be given in such a way that the input buffer can only be removed integers from its beginning and the output buffer can only be appended integers to its end. If there is no integer left in the input buffer then `read()` blocks. The output buffer is assumed unbounded, so `print(`a`)` never blocks when outputting the value of a. Consider the following two programs:

```
var m, n, s ;
n := read() ;
while (m <= n) do
  ( print(m) ;  s := s + m ;  m := m + 1 ) ;
print(s)
```

```
var s ;
s := 0 ;
while not(read() <= 0) do
  s := s + read() ;
print(s)
```

The first reads one integer i from the beginning of the input buffer and then it appends $i+2$ integers to the end of the output buffer (the numbers 0, 1, 2, ..., i followed by their sum). The second reads a potentially unbounded number of integers from the input buffer, terminating if and only if it reads a non-positive integer on an odd position in the input buffer; when that happens, it outputs the sum of the elements on the even positions in the input buffer up to that point.

It is interesting to note that the addition of `read()` to IMP means that expression evaluation becomes non-deterministic, regardless of whether we have variable increment or not in our language. Indeed, since `/` is non-deterministic, an expression of the form `read() / read()` can evaluate the two reads in any order; for example, if the first two integers in the input buffer are 7 and 3, then this expression can evaluate to either 2 or 0.

Let us formalize buffers. Assume colon-separated integer lists with ϵ as identity, $\mathbf{List}^{\epsilon}_{:}\{Int\}$, and let ω, ω', ω_1, etc., range over such lists of integers. The same way we decided for notational convenience to let *State* be an alias for the map sort $\mathbf{Map}\{Id \mapsto Int\}$ (Section 3.1.2), from here on we also let *Buffer* alias the list sort $\mathbf{List}^{\epsilon}_{:}\{Int\}$.

In a formal language semantics, providing the entire input as part of the initial configuration and collecting the entire output in the result configuration is acceptable, although in implementations of the language one will most likely want the input/output to be interactive. There is some flexibility as to where the input and output buffers should be located in the configuration. One possibility is as new top-level components in the configuration. Another possibility is as special variables in the already existing state. The latter would require some non-trivial changes in the mathematical model of the state, so we prefer to follow the former approach in the sequel. An additional argument in favor of our choice is that sooner or later one needs to add new components to the configuration anyway, so we take this opportunity to discuss how robust/modular the various semantic styles are with regards to changes in the structure of the configuration.

Big-Step SOS

To accommodate the input and the output buffers, all configurations and all sequents we had in the original big-step SOS of IMP in Sections 3.2.1 and 3.2.2 need to change. For example, since expressions can now consume input, the original expression sequents of form $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t\rangle$ need to change into sequents $\langle a, \sigma, \omega\rangle \Downarrow \langle i, \omega'\rangle$ and $\langle b, \sigma, \omega\rangle \Downarrow \langle t, \omega'\rangle$ (recall that we add one feature at a time, so expression evaluation currently does not have side effects on the state), respectively, where $\omega, \omega' \in Buffer$. Also, the big-step SOS rules for expressions need to change to take into account both the new configurations and the fact that expression evaluation can now affect the input buffer. For example, the original big-step SOS of division, namely

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ / } a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int}\, i_2\rangle} \quad \text{if } i_2 \neq 0$$

changes as follows:

$$\frac{\langle a_1, \sigma, \omega\rangle \Downarrow \langle i_1, \omega_1\rangle \qquad \langle a_2, \sigma, \omega_1\rangle \Downarrow \langle i_2, \omega_2\rangle}{\langle a_1 \texttt{ / } a_2, \sigma, \omega\rangle \Downarrow \langle i_1 /_{Int} i_2, \omega_2\rangle}\,, \quad \text{where } i_2 \neq 0$$

$$\frac{\langle a_1, \sigma, \omega_2\rangle \Downarrow \langle i_1, \omega_1\rangle \qquad \langle a_2, \sigma, \omega\rangle \Downarrow \langle i_2, \omega_2\rangle}{\langle a_1 \texttt{ / } a_2, \sigma, \omega\rangle \Downarrow \langle i_1 /_{Int} i_2, \omega_1\rangle}\,, \quad \text{where } i_2 \neq 0$$

Like for the variable increment, the rules above make an attempt to capture the intended nondeterministic evaluation strategy of the division operator. Unfortunately, they also only capture a non-deterministic choice strategy, failing to capture the intended full non-determinism of division.

Since statements can both consume input and produce output, their big-step SOS sequents need to change from $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ to $\langle s, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma', \omega'_{in}, \omega_{out}\rangle$, where $\omega_{in}, \omega'_{in} \in Buffer$ are the input buffers before and, respectively, after the evaluation of statement s, and where $\omega_{out} \in \mathit{Buffer}$ is the output produced during the evaluation of s. Unfortunately, all big-step SOS rules for statements also have to change, to accommodate the additional input and/or output components in configurations. For example, the semantics of sequential composition needs to change from

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \sigma_1\rangle \qquad \langle s_2, \sigma_1\rangle \Downarrow \langle \sigma_2\rangle}{\langle s_1 \,;\, s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle}$$

to

$$\frac{\langle s_1, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma_1, \omega_{in}^1, \omega_{out}^1\rangle \qquad \langle s_2, \sigma_1, \omega_{in}^1\rangle \Downarrow \langle \sigma_2, \omega_{in}^2, \omega_{out}^2\rangle}{\langle s_1 \,;\, s_2, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma_2, \omega_{in}^2, \omega_{out}^1 : \omega_{out}^2\rangle}$$

Note that the outputs of s_1 and of s_2 have been appended in order to yield the output of s_1 ; s_2.

Finally, we also have to change the initial configuration holding programs to also take an input, as well as the big-step SOS rule for programs from

$$\frac{\langle s, \mathit{xl} \mapsto 0\rangle \Downarrow \langle \sigma\rangle}{\langle\, \texttt{var}\ \mathit{xl}\ ;\ s\rangle \Downarrow \langle \sigma\rangle}$$

into

$$\frac{\langle s, \mathit{xl} \mapsto 0, \omega_{in}\rangle \Downarrow \langle \sigma, \omega_{in}', \omega_{out}\rangle}{\langle\, \texttt{var}\ \mathit{xl}\ ;\ s, \omega_{in}\rangle \Downarrow \langle \sigma, \omega_{in}', \omega_{out}\rangle}$$

We preferred to also keep the input buffer in the final configuration for two reasons: to avoid having to define a new configuration holding only the state and the output, and to allow the language designer to more easily debug her semantics, in particular to see whether there is any input left unused at the end of the program. One can argue that now, since we have output in our language, the result configuration may actually contain only the output (that is, it can also drop the state). The reader is encouraged to experiment with different final configurations.

Hence, almost everything changed in the original big-step SOS of IMP in order to prepare for the addition of the input/output constructs. All theses necessary changes highlight, again, the lack of modularity of big-step SOS. Once all the changes above are applied, one can easily define the semantics of the input/output constructs as follows:

$$\langle \texttt{read()}, \sigma, i : \omega_{in}\rangle \Downarrow \langle i, \omega_{in}\rangle \qquad \text{(BigStep-Read)}$$

$$\frac{\langle a, \sigma, \omega_{in}\rangle \Downarrow \langle i, \omega_{in}'\rangle}{\langle \texttt{print(}\,a\,\texttt{)}, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma, \omega_{in}', i\rangle} \qquad \text{(BigStep-Print)}$$

Type System using Big-Step SOS

The typing policy of the input/output constructs is straightforward, though recall that we decided to only allow to read and to print integers. To type programs using input/output, we therefore add the following typing rules to the already existing typing rules in Figure 3.10:

$$\mathit{xl} \vdash \texttt{read()} : \mathit{int} \qquad \text{(BigStepTypeSystem-Read)}$$

$$\frac{\mathit{xl} \vdash a : \mathit{int}}{\mathit{xl} \vdash \texttt{print(}\,a\,\texttt{)} : \mathit{stmt}} \qquad \text{(BigStepTypeSystem-Print)}$$

Small-Step SOS

Like in big-step SOS, in small-step SOS we also need to change all IMP's configurations in Section 3.3.1 in order to add input/output. In the spirit of making only minimal changes, we modify the configurations holding expressions to also only hold an input buffer, and the configurations holding statements to also hold both an input and an output buffer. Implicitly, all IMP's small-step SOS rules in Section 3.3.2 also need to change. The changes are straightforward, essentially having to just propagate the output through each statement construct, but they are still changes and thus expose, again, the lack of modularity of small-step SOS. Here is, for example, how the small-step SOS rules for division and for sequential composition need to change:

$$\frac{\langle a_1, \sigma, \omega_{in} \rangle \to \langle a_1', \sigma, \omega_{in}' \rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in} \rangle \to \langle a_1' \,/\, a_2, \sigma, \omega_{in}' \rangle}$$

$$\frac{\langle a_2, \sigma, \omega_{in} \rangle \to \langle a_2', \sigma, \omega_{in}' \rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in} \rangle \to \langle a_1 \,/\, a_2', \sigma, \omega_{in}' \rangle}$$

$$\frac{\langle s_1, \sigma, \omega_{in}, \omega_{out} \rangle \to \langle s_1', \sigma', \omega_{in}', \omega_{out}' \rangle}{\langle s_1 \,;\, s_2, \sigma, \omega_{in}, \omega_{out} \rangle \to \langle s_1' \,;\, s_2, \sigma', \omega_{in}', \omega_{out}' \rangle}$$

The expression configurations do not need to consider an output buffer because, as already discussed, in this language design experiment we assume at each stage only the current feature, without attempting to anticipate other features that will be possibly added in the future, and we attempts to do minimal changes. For example, if functions were to be added to the language later, in which case expressions will also possibly affect the output through function calls, then all the expression configurations and their corresponding small-step SOS rules will need to change again.

Finally, we also have to change the initial configuration holding programs to also take an input, as well as the small-step SOS rule for programs to initialize the output buffer to ϵ as follows:

$$\langle \, \texttt{var} \, xl \,;\, s, \omega_{in} \rangle \to \langle s, (xl \mapsto 0), \omega_{in}, \epsilon \rangle$$

Once all the changes are applied, we can give the small-step SOS of input/output as follows:

$$\langle \, \texttt{read()}, \sigma, i : \omega_{in} \rangle \to \langle i, \sigma, \omega_{in} \rangle \qquad (\textsc{SmallStep-Read})$$

$$\frac{\langle a, \sigma, \omega_{in} \rangle \to \langle a', \sigma, \omega_{in}' \rangle}{\langle \, \texttt{print(} \, a \, \texttt{)}, \sigma, \omega_{in}, \omega_{out} \rangle \to \langle \, \texttt{print(} \, a' \, \texttt{)}, \sigma, \omega_{in}', \omega_{out} \rangle} \qquad (\textsc{SmallStep-Print-Arg})$$

$$\langle \, \texttt{print(} \, i \, \texttt{)}, \sigma, \omega_{in}, \omega_{out} \rangle \to \langle \, \texttt{skip} \, , \sigma, \omega_{in}, \omega_{out} : i \rangle \qquad (\textsc{SmallStep-Print})$$

Denotational Semantics

To accommodate the input and the output buffers, the denotation functions associated to IMP's syntactic categories need to change their types from

$$\begin{array}{l} [\![\_]\!] : \mathit{AExp} \to (\mathit{State} \rightharpoonup \mathit{Int}) \\ [\![\_]\!] : \mathit{BExp} \to (\mathit{State} \rightharpoonup \mathit{Bool}) \\ [\![\_]\!] : \mathit{Stmt} \to (\mathit{State} \rightharpoonup \mathit{State}) \\ [\![\_]\!] : \mathit{Pgm} \to \mathit{State}_\bot \end{array}$$

to

$$\begin{array}{l}
[\![\_]\!] : AExp \to (State \times Buffer \rightharpoonup Int \times Buffer)\\
[\![\_]\!] : BExp \to (State \times Buffer \rightharpoonup Bool \times Buffer)\\
[\![\_]\!] : Stmt \to (State \times Buffer \rightharpoonup State \times Buffer \times Buffer)\\
[\![\_]\!] : Pgm \to (Buffer \rightharpoonup State \times Buffer \times Buffer)
\end{array}$$

We next briefly explain the definitions of the new denotation functions.

The denotations of expressions now take a state and an input buffer and produce a value and a possibly modified input buffer (since `read()` may consume elements from the input buffer). For example, the denotation of division becomes:

$$[\![a_1 \,/\, a_2]\!]\pi = \begin{cases} (1^{\text{st}}(arg_1) \;/_{Int}\; 1^{\text{st}}(arg_2),\; 2^{\text{nd}}(arg_2)) & \text{if } 1^{\text{st}}(arg_2) \neq 0 \\ \bot & \text{if } 1^{\text{st}}(arg_2) = 0 \end{cases}$$

where $arg_1 = [\![a_1]\!]\pi$ and $arg_2 = [\![a_2]\!](1^{\text{st}}(\pi), 2^{\text{nd}}(arg_1))$.

The denotations of statements can now produce an output buffer in addition to a modified input buffer (and a state). For example, the denotation of sequential composition becomes:

$$[\![s_1 \,;\, s_2]\!]\pi = (1^{\text{st}}(arg_2),\; 2^{\text{nd}}(arg_2),\; 3^{\text{rd}}(arg_1) : 3^{\text{rd}}(arg_2))$$

where $arg_1 = [\![s_1]\!]\pi$ and $arg_2 = [\![s_2]\!](1^{\text{st}}(arg_1),\; 2^{\text{nd}}(arg_1))$. As another example of statement denotational semantics, the denotational semantics of while loops $\texttt{while}\, b\, \texttt{do}\, s$ remains a fixed-point, but in order to be consistent with the new type of the denotation function for statements, it needs to be the fixed point of a (total) function of the form

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

It is not difficult to see that the following definition of $\mathcal{F}$ has the right type and that $\mathcal{F}(\alpha)$ indeed captures the information that is added to α by unrolling the loop once:

$$\mathcal{F}(\alpha)(\pi) = \begin{cases} (1^{\text{st}}(arg_3),\; 2^{nd}(arg_3),\; 3^{\text{rd}}(arg_2) : 3^{\text{rd}}(arg_3)) & \text{if } 1^{\text{st}}(arg_1) = \texttt{true} \\ (1^{\text{st}}(\pi),\; 2^{\text{nd}}(arg_1),\; \epsilon) & \text{if } 1^{\text{st}}(arg_1) = \texttt{false} \\ \bot & \text{if } arg_1 = \bot \end{cases}$$

where $arg_1 = [\![b]\!]\pi$, $arg_2 = [\![s]\!](1^{\text{st}}(\pi),\; 2^{\text{nd}}(arg_1))$, and $arg_3 = \alpha(1^{\text{st}}(arg_2),\; 2^{\text{nd}}(arg_2))$.

Programs now take an input as well; like for big-step SOS, we prefer to also make the remaining input available at the end of the program execution, in addition to the state and the output:

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

Once all the changes on the denotations of the various syntactic categories are applied as discussed above, adding the semantics of the new input/output constructs is immediate:

$$[\![\texttt{read()}]\!]\pi = (head(2^{\text{nd}}(\pi)),\; tail(2^{\text{nd}}(\pi))) \qquad \text{(DENOTATIONAL-Read)}$$

$$[\![\texttt{print(}\, a \,\texttt{)}]\!]\pi = (1^{\text{st}}(\pi),\; 2^{\text{nd}}([\![a]\!]\pi),\; 1^{\text{st}}([\![a]\!]\pi)) \qquad \text{(DENOTATIONAL-Print)}$$

Like in the case of IMP's extension with the variable increment expression construct, the denotational semantics of the extension with the input/output constructs would have been more modular if we had adopted a continuation or monadic style from the very beginning.

3.5.3 Adding Abrupt Termination

IMP++ adds both implicit and explicit abrupt program termination. The implicit abrupt termination is given by division by zero, while the explicit abrupt termination is given by a new statement added to the language, `halt`.

For the sake of making a choice and also for demonstration purposes, in both cases of abrupt termination we would like, admittedly subjectively, the resulting configuration to have the same structure as if the program terminated normally; for example, in the case of big-step SOS, we would like the result configuration for statements to be $\langle\sigma\rangle$, where σ is the state when the program was terminated abruptly. For example, we want the programs

```
var m, n, s ;
n := 100 ;
while true do
  if m <= n
  then (
         s := s + m ;
         m := m + 1
       )
  else halt
```

```
var m, n, s ;
n := 100 ;
while true do
  if m <= n
  then (
         s := s + m ;
         m := m + 1
       )
  else s := s / (n / m)
```

to yield the result configuration `< m |-> 101 & n |-> 100 & s |-> 5050 >` instead of a special configuration of the form `< halting, m |-> 101 & n |-> 100 & s |-> 5050 >` or similar. Unfortunately, that is not possible in all cases without intrusively modifying the syntax of the IMP language (to catch the exceptional behavior and explicitly discard the additional information), since some operational styles need to make a sharp distinction between a halting configuration and a normal configuration (for propagation reasons). Proponents of those semantic styles may argue that our semantic choice above seems inappropriate, since giving more information in the result configuration, such as "this is a halting configuration", is better for all purposes than giving less information. There are, however, also reasons to always want a normal result configuration upon termination. For example, one may want to include IMP in a larger context, such as in a distributed system, where all the context wants to know about the embedded language is that it takes a statement and produces a state and/or an output; IMP's internal exceptional situations are of no concern to the outer context. There is no absolute better or worse language design, both in what regards syntax and in what regards semantics. Our task here is to make the language designer aware of the subtleties and the limitations of the various semantic approaches.

Big-Step SOS

The lack of modularity of big-step semantics will be, again, emphasized here. Let us first add the semantic definition for the implicit abrupt termination generated by division by zero. Recall that the big-step SOS rule for division was the following:

$$\frac{\langle a_1,\sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2,\sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ / } a_2,\sigma\rangle \Downarrow \langle i_1 /_{\mathit{Int}}\, i_2\rangle}\text{, where } i_2 \neq 0$$

We keep that unchanged, but we also add the following new rule:

$$\frac{\langle a_1,\sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2,\sigma\rangle \Downarrow \langle 0\rangle}{\langle a_1 \texttt{ / } a_2,\sigma\rangle \Downarrow \langle \texttt{error}\rangle} \qquad \text{(BigStep-Div-By-Zero)}$$

In the above rule, `error` can be regarded as a special value; alternatively, one can regard $\langle \texttt{error} \rangle$ as a special result configuration.

But what if the evaluation of a_1 or of a_2 in the above rule generates itself an error? If that is the case, then one needs to propagate that error through the division construct:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle \texttt{error}\rangle}{\langle a_1 \texttt{ / } a_2, \sigma\rangle \Downarrow \langle \texttt{error}\rangle} \qquad (\textsc{BigStep-Div-Error-Left})$$

$$\frac{\langle a_2, \sigma\rangle \Downarrow \langle \texttt{error}\rangle}{\langle a_1 \texttt{ / } a_2, \sigma\rangle \Downarrow \langle \texttt{error}\rangle} \qquad (\textsc{BigStep-Div-Error-Right})$$

Note that in case one of a_1 or a_2 generates an error, then the other one is not even evaluated anymore, to faithfully capture the intended meaning of abrupt termination.

Unfortunately, one has to do the same for all the expression language constructs. This way, for each expression construct, one has to add at least as many error-propagation big-step SOS rules as arguments that expression construct takes. Moreover, when the evaluation error reaches a statement, one needs to transform it into a "halting signal". This can be achieved by introducing a new type of result configuration, namely $\langle \texttt{halting}, \sigma\rangle$, and then adding appropriate halting propagation rules for all the statements. For example, the assignment statement needs to be added the new rule

$$\frac{\langle a, \sigma\rangle \Downarrow \langle \texttt{error}\rangle}{\langle x \texttt{ := } a, \sigma\rangle \Downarrow \langle \texttt{halting}, \sigma\rangle} \qquad (\textsc{BigStep-Asgn-Halt})$$

The halting signal needs to be propagated through statement constructs, collecting the appropriate state. For example, the following two rules need to be included for sequential composition, in addition to the existing rule (which stays unchanged):

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \texttt{halting}, \sigma_1\rangle}{\langle s_1 \texttt{;} s_2, \sigma\rangle \Downarrow \langle \texttt{halting}, \sigma_1\rangle} \qquad (\textsc{BigStep-Seq-Halt-Left})$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \sigma_1\rangle,\ \langle s_2, \sigma_1\rangle \Downarrow \langle \texttt{halting}, \sigma_2\rangle}{\langle s_1 \texttt{ ; } s_2, \sigma\rangle \Downarrow \langle \texttt{ halting }, \sigma_2\rangle} \qquad (\textsc{BigStep-Seq-Halt-Right})$$

In addition to all the halting propagation rules, we also have to define the semantics of the explicit halt statement:

$$\langle \texttt{ halt }, \sigma\rangle \Downarrow \langle \texttt{ halting }, \sigma\rangle \qquad (\textsc{BigStep-Halt})$$

Therefore, when using big-step SOS, one has to more than *double* the number of rules in order to support abrupt termination. Indeed, any argument of any language construct can yield the termination signal, so a rule is necessary to propagate that signal through the current language construct. It is hard to imagine anything worse in a language design framework. An unfortunate language designer choosing big-step semantics as her favorite language definition framework will incrementally become very reluctant to add or experiment with any new feature in her language. For example, imagine that one wants to add exceptions and break/continue of loops to IMP++.

Finally, unless one extends the language syntax, there appears to be no way to get rid of the junk result configurations $\langle \texttt{ halting }, \sigma\rangle$ that have been artificially added in order to propagate the error or the halting signals. For example, one cannot simply add the rule

$$\frac{\langle s, \sigma \rangle \Downarrow \langle \texttt{halting}, \sigma' \rangle}{\langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle}$$

because it may interfere with other rules and thus wrongly hide the halting signal; for example, it can be applied on the second hypothesis of the rule (BigStep-Seq-Halt-Right) above hiding the halting signal and thus wrongly making the normal rule (BigStep-Seq) applicable. While having junk result configurations of the form $\langle \texttt{halting}, \sigma \rangle$ may seem acceptable in our scenario here, perhaps even desirable for debugging reasons, in general one may find it inconvenient to have many types of result configurations; indeed, one would need similar junk configurations for exceptions, for break/continue of loops, for functions return, etc.

Consequently, the halting signal needs to be caught at the top-level of the derivation. Fortunately, IMP provides a top-level syntactic category, *Pgm*, so we can add the following rule which dissolves the potential junk configuration at the top:

$$\frac{\langle s, xl \mapsto 0 \rangle \Downarrow \langle \texttt{halting}, \sigma \rangle}{\langle \texttt{var}\ xl\ ;\ s \rangle \Downarrow \langle \sigma \rangle} \qquad \text{(BigStep-Halt)}$$

Now we have silent abrupt termination. Note that `var` now acts as an exception catching and dissolving the abrupt termination signal generated by `halt` or by division-by-zero. If one does not like to use `var` for something which has not been originally intended, then one can add an auxiliary `top` statement or program construct, then reduce the semantics of `var` to that of `top`, and then give `top` an exception-handling-like big-step SOS, as we will do for the MSOS definition of abrupt termination in Section 3.6; see Exercise 75. This latter solution is also more general, because it does not rely on a fortunate earlier decision to have a top-level language construct.

In addition to the lack of modularity due to having to more than double the number of rules in order to add abrupt termination, the inclusion of all these rules can also have a significant impact on performance when one wants to execute the big-step SOS. Indeed, there are now four rules for division, each having the same left-hand side, $\langle a_1 \,\texttt{/}\, a_2, \sigma \rangle$, and some of these rules even sharing some of the hypotheses. That means that any general-purpose proof or rewrite system attempting to execute such a definition will unavoidably face the problem of searching a large space of possibilities in order to find one or all possible reductions.

Type System using Big-Step SOS

The typing policy of abrupt termination is clear: `halt` types to a statement and division-by-zero is ignored. Indeed, one cannot expect that a type checker, or any technique, procedure or algorithm, can detect division by zero in general: division-by-zero, like almost any other runtime property of any Turing-complete programing language, is an undecidable problem. Consequently, it is common to limit typing division to checking that the two expressions have the expected type, integer in our case, which our existing type checker for IMP already does (see Figure 3.10). We therefore only add the following typing rule for `halt`:

$$xl \vdash \texttt{halt} : stmt \qquad \text{(BigStepTypeSystem-Halt)}$$

Small-Step SOS

Small-step SOS turns out to be almost as non-modular as big-step SOS when defining control-intensive constructs like abrupt termination. Like for big-step SOS, we need to invent special configurations to signal steps corresponding to implicit division by zero or to explicit halt statements. In small-step SOS, a single type of such special configurations suffices, namely one of the form $\langle\,\texttt{halting}\,,\sigma\rangle$, where σ is the state in which the program was abruptly terminated. However, for uniformity with big-step SOS, we also define two types of special configurations, one for expressions and one for statements; since we decided to carry the state in the right-hand-side configuration of all sequents in our small-step SOS definitions, the only difference between the two types of configurations is their tag, namely $\langle\,\texttt{error}\,,\sigma\rangle$ for the former versus $\langle\,\texttt{halting}\,,\sigma\rangle$ for the latter.

We can then define the small-step SOS of division by zero as follows (recall that the original SmallStep-Div rule in Figure 3.14 is "$\langle i_1\ \texttt{/}\ i_2,\sigma\rangle \to \langle i_1 /_{Int}\, i_2,\sigma\rangle$ where $i_2 \neq 0$"):

$$\langle i_1\ \texttt{/}\ 0,\sigma\rangle \to \langle\texttt{error}\,,\sigma\rangle \qquad \text{(SmallStep-Div-By-Zero)}$$

Like for the big-step SOS extension above, we have to make sure that the halting signal is correctly propagated. Here are, for example, the propagation rules through the division construct:

$$\frac{\langle a_1,\sigma\rangle \to \langle\texttt{error}\,,\sigma\rangle}{\langle a_1\,\texttt{/}\,a_2,\sigma\rangle \to \langle\texttt{error}\,,\sigma\rangle} \qquad \text{(SmallStep-Div-Error-Left)}$$

$$\frac{\langle a_2,\sigma\rangle \to \langle\texttt{error}\,,\sigma\rangle}{\langle a_1\,\texttt{/}\,a_2,\sigma\rangle \to \langle\texttt{error}\,,\sigma\rangle} \qquad \text{(SmallStep-Div-Error-Right)}$$

The two rules above are given in such a way that the semantics is faithful to the intended *computational granularity* of the defined language feature. Indeed, we want division by zero to take one computational step to be reported as an error, as opposed to as many steps as the depth of the context in which the error has been detected; for example, a configuration containing expression `(3 / 0) / 3` should reduce to a halting configurations in one step, not in two. If one added a special error integer value and replaced the two rules above by

$$\langle\,\texttt{error}\,\texttt{/}\,a_2,\sigma\rangle \to \langle\,\texttt{error}\,,\sigma\rangle$$

$$\langle a_1\,\texttt{/}\,\texttt{error}\,,\sigma\rangle \to \langle\,\texttt{error}\,,\sigma\rangle$$

then errors would be propagated to the top level of the program in as many small-steps as the depth of the context in which the error was generated; we do not want that.

Like in the big-step SOS above, the implicit expression errors need to propagate through the statements and halt the program. One way to do it is to generate an explicit `halt` statement and then to propagate that `halt` statement through all the statement constructs as if it was a special statement value, until it reaches the top. However, as discussed in the paragraph above, that would generate as many steps as the depth of the evaluation contexts in which the `halt` statement is located, instead of just one step as desired. Alternatively, we can use the same approach to propagate the halting configuration through the statement constructs as we used to propagate it through the expression constructs. More precisely, we add transition rules from expressions to statements, like the one below (a similar one needs to be added for the conditional statement):

$$\frac{\langle a,\sigma\rangle \to \langle\texttt{error}\,,\sigma\rangle}{\langle x \;\texttt{:=}\; a,\sigma\rangle \to \langle\texttt{halting}\,,\sigma\rangle} \qquad \text{(SmallStep-Asgn-Halt)}$$

Once the halting signal due to a division by zero reaches the statement level, it needs to be further propagated through the sequential composition, that is, we need to add the following rule:

$$\frac{\langle s_1, \sigma\rangle \rightarrow \langle \texttt{halting}, \sigma\rangle}{\langle s_1 \,;\, s_2, \sigma\rangle \rightarrow \langle \texttt{halting}, \sigma\rangle} \qquad (\textsc{SmallStep-Seq-Halt})$$

Note that we assumed that a halting step does not change the state (used the same σ in both the left and the right configurations). One can prove by structural induction that, in our simple language scenario, that is indeed the case; alternatively, one could have equivalently used σ' instead of σ in the right-hand configurations in the rule above.

Finally, we can also generate a special halting configuration when a `halt` statement is reached:

$$\langle \texttt{halt}, \sigma\rangle \rightarrow \langle \texttt{halting}, \sigma\rangle \qquad (\textsc{SmallStep-Halt})$$

At this moment, any abruptly terminated program reduces to a special configuration of the form $\langle \texttt{halting}, \sigma\rangle$. Recall that our plan, however, was to terminate the computation with a normal configuration of the form $\langle \texttt{skip}, \sigma\rangle$, regardless of whether the program terminates normally or abruptly. Like in the big-step SOS above, the naive solution to transform a step producing a halting configuration into a normal step using the rule

$$\frac{\langle s, \sigma\rangle \rightarrow \langle \texttt{halting}, \sigma\rangle}{\langle s, \sigma\rangle \rightarrow \langle \texttt{skip}, \sigma\rangle}$$

does not work. Indeed, consider a situation where the rule (SmallStep-Seq-Halt) above could apply. There is nothing to prevent the naive rule above to interfere and transform the halting premise of (SmallStep-Seq-Halt) into a normal step producing a `skip`, which can be further fed to the conventional rule for sequential composition, (SmallStep-Seq-Arg1) in Figure 3.15, hereby continuing the execution of the program as if no abrupt termination took place.

Supposing that one wants to waste no computational steps as an artifact of the particular small-step SOS approach chosen, there seems to be no immediate way to terminate the program with a normal result configuration of the form $\langle \texttt{skip}, \sigma\rangle$ both when the program terminates abruptly and when it terminates normally. One possibility, also suggested in the case of big-step SOS discussed above and followed in the subsequent MSOS definition for abrupt termination in Section 3.6, is to add an auxiliary `top` language construct. With that, we can change the small-step SOS rule for variable declarations to reduce the top-level program to its body statement wrapped under this `top` construct, and then add corresponding rules to propagate normal steps under the `top` while catching the halting steps and transforming them into normal steps at the top-level. Here are four small-step SOS rules achieving this (the first rule replaces the previous one for variable declarations); see also Exercise 78:

$$\langle \texttt{var}\, xl \,;\, s\rangle \rightarrow \langle \texttt{top}\, s, (xl \mapsto 0)\rangle$$

$$\frac{\langle s, \sigma\rangle \rightarrow \langle s', \sigma'\rangle}{\langle \texttt{top}\, s, \sigma\rangle \rightarrow \langle \texttt{top}\, s', \sigma'\rangle}$$

$$\langle \texttt{top skip}, \sigma\rangle \rightarrow \langle \texttt{skip}, \sigma\rangle$$

$$\frac{\langle s, \sigma\rangle \rightarrow \langle \texttt{halting}, \sigma\rangle}{\langle \texttt{top}\, s, \sigma\rangle \rightarrow \langle \texttt{skip}, \sigma\rangle}$$

The rules above are such that no computational step is wasted when a halting signal takes place. Unfortunately, we had to introduce additional syntax (the `top` construct) for the sole purpose of making the semantic definition work. If one is willing to waste a computational step in order to explicitly dissolve the halting configuration replacing it by a normal one, then one can add instead the following simple small-step SOS rule, which is also the approach we are taking in the case of small-step SOS in this section, because it gives us, in our view, the best trade-off between elegance and computational intrusiveness (after all, wasting a step in a deterministic manner may be acceptable in most situations):

$$\langle \texttt{halting}, \sigma \rangle \rightarrow \langle \texttt{skip}, \sigma \rangle \qquad \text{(SmallStep-Halting)}$$

Denotational Semantics

Since both expressions and statements can abruptly terminate, like in the previous semantics we have to provide a means for the denotation functions for expressions and statements to flag when abrupt termination is intended. This way, the denotation of programs can catch the abrupt termination flag and yield the expected state (recall that we want to see normal termination of programs regardless of whether that happens abruptly or not). Specifically, we change the previous denotation functions

$$\begin{array}{l} [\![\_]\!] : \mathit{AExp} \rightarrow (\mathit{State} \rightharpoondown \mathit{Int}) \\ [\![\_]\!] : \mathit{BExp} \rightarrow (\mathit{State} \rightharpoondown \mathit{Bool}) \\ [\![\_]\!] : \mathit{Stmt} \rightarrow (\mathit{State} \rightharpoondown \mathit{State}) \end{array}$$

into denotation functions of the form

$$\begin{array}{l} [\![\_]\!] : \mathit{AExp} \rightarrow (\mathit{State} \rightharpoondown \mathit{Int} \cup \{\mathit{error}\}) \\ [\![\_]\!] : \mathit{BExp} \rightarrow (\mathit{State} \rightharpoondown \mathit{Bool} \cup \{\mathit{error}\}) \\ [\![\_]\!] : \mathit{Stmt} \rightarrow (\mathit{State} \rightharpoondown \mathit{State} \times \{\mathit{halting}, \mathit{ok}\}) \end{array}$$

as described below. Before we proceed, note that the new denotation functions will still associate partial functions to syntactic categories. While the new semantics will be indeed able now to catch and terminate when a division by zero takes place, it will still not be able to catch non-termination of programs; the denotation of those programs will still stay undefined. Strictly technically speaking, the denotations of expressions will now always be defined, because the only source of expression undefinedness in IMP was division by zero. However, for the same reason it is good practice in small-step SOS to have the same type of configurations both to the left and to the right of the transition arrow ($\rightarrow$), it is good practice in denotational semantics to work with domains of partial functions instead of ones of total functions. This way, we wouldn't have to change these later on if we add new expression constructs to the language that yield undefinedness (such as, e.g., functions).

The denotation functions of expressions need to change in order to initiate a "catchable" error when division by zero takes place, and then to propagate it through all the other expression constructs. We only discuss the denotation of division, the other being simpler. The previous denotation function of division $[\![a_1 \,/\, a_2]\!]$ was defined as

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

which is not good enough anymore. To catch and propagate the division-by-zero error, we can modify the denotation of division as follows:

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

The second case above is necessary, because we want $[\![a_1 \,/\, a_2]\!]\sigma$ to be undefined, and not *error*, when $[\![a_1]\!]\sigma = \bot$ and $[\![a_2]\!]\sigma = 0$.

Like in the previous semantics, the implicit expression errors need to propagate through the statements and halt the program. The denotation function of statements now returns both a state and a flag. The flag tells whether the state resulted from a normal evaluation or whether it is a halting state that needs to be propagated. Here is the new denotation of assignment:

$$[\![x := a]\!]\sigma = \begin{cases} (\sigma, \mathit{halting}) & \text{if } [\![a]\!]\sigma = \mathit{error} \\ \bot & \text{if } \sigma(x) = \bot \text{ or } [\![a]\!]\sigma = \bot \\ (\sigma[[\![a]\!]\sigma/x], \mathit{ok}) & \text{if otherwise} \end{cases}$$

Above, we chose to halt when $[\![a]\!]\sigma = \mathit{error}$ and $\sigma(x) = \bot$ (the cases are handled in order). The alternative would be to choose undefined instead of halt (see Exercise 81). Our choice to assume that the expression being assigned is always evaluated no matter whether the assigned variable is declared or not, is consistent with the small-step SOS of assignment in IMP: first evaluate the expression being assigned step by step, then write the resulting value to the assigned variable if declared; if undeclared then get stuck (Section 3.3.2). The statement sequential composition construct needs to also properly propagate the halting situation:

$$[\![s_1 \; ; \; s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if } 2^{\text{nd}}([\![s_1]\!]\sigma) = \mathit{halting} \\ [\![s_2]\!](1^{\text{st}}([\![s_1]\!]\sigma)) & \text{if } 2^{\text{nd}}([\![s_1]\!]\sigma) = \mathit{ok} \\ \bot & \text{if } [\![s_1]\!]\sigma = \bot \end{cases}$$

We also discuss the denotational semantics of while loops `while` b `do` s. It now needs to be the fixed point of a (total) function of the form

$$\mathcal{F} : (\mathit{State} \rightharpoonup \mathit{State} \times \{\mathit{halting}, \mathit{ok}\}) \to (\mathit{State} \rightharpoonup \mathit{State} \times \{\mathit{halting}, \mathit{ok}\})$$

The following defines such an $\mathcal{F}$ which has the right type and captures the information that is added by unrolling the loop once (the cases are handled in order, from top to bottom):

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \bot & \text{if } [\![b]\!]\sigma = \bot \\ (\sigma, \mathit{halting}) & \text{if } [\![b]\!]\sigma = \mathit{error} \\ (\sigma, \mathit{ok}) & \text{if } [\![b]\!]\sigma = \texttt{false} \\ [\![s]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \text{ and } 2^{\text{nd}}([\![s]\!]\sigma) = \mathit{halting} \\ \alpha(1^{\text{st}}([\![s]\!]\sigma)) & \text{if } [\![b]\!]\sigma = \texttt{true} \text{ and } 2^{\text{nd}}([\![s]\!]\sigma) = \mathit{ok} \end{cases}$$

After we modify the denotation functions of almost all expression and statement constructs as explained above (except for the denotations of variable lookup, and of builtin integers and booleans), we have to also modify the denotation of programs to silently discard the halting signal in case its body statement terminated abruptly (the type of the denotation of programs does not change):

$$[\![\texttt{var}\, \mathit{xl} \; ; \; s]\!] = 1^{\text{st}}([\![s]\!](\mathit{xl} \mapsto 0))$$

Finally, we can now also give the denotational semantics of `halt`:

$$[\![\texttt{halt}]\!]\sigma = (\sigma, \mathit{halting})$$

Like for the other semantic extensions in this section, adding the semantics of abrupt termination is easy; the tedious part is to modify the existing semantics to make it aware of abrupt termination.

3.5.4 Adding Dynamic Threads

IMP++ adds threads to IMP, which can be dynamically created and terminated. As in the previous IMP extensions, we keep the syntax and the semantics of threads minimal. Our only syntactic extension is a `spawn` statement construct. The semantics of `spawn` S is that the statement S executes concurrently with the rest of the program, sharing and possibly concurrently modifying the same variables, like threads do. The thread corresponding to S terminates when S terminates and, when that happens, we remove the thread. Like in the previous language extensions, we want programs to terminate normally, no matter whether they make use of threads or not. For example, the programs below create 101 and, respectively, 2 new threads during their execution:

```
var m, n, s ;
n := 100 ;
while (m <= n) do (
  spawn s := s + m ;
  m := m + 1
)
```

```
var x ;
(
   spawn x := x + 1 ;
   spawn x := x + 10
) ;
x := x + 100
```

Yet, we want the result configurations at the end of the execution to look like before in IMP, that is, like `< skip, m |-> 101 & n |-> 100 & s |-> 5050 >` and `< skip, x |-> 111 >`, respectively, in the case of small-step SOS. We grouped the two `spawn` statements in the second program on purpose, to highlight the need for structural equivalences (this will be discussed shortly, in the context of small-step SOS). Recall that the syntax of IMP's sequential composition in Section 3.1 (see Figure 3.1) was deliberately left ambiguous, based on the hypothesis that the semantics of IMP will be given in such a way (left-to-right statement evaluation) that the syntactic ambiguity is irrelevant. Unfortunately, the addition of threads makes the above hard or impossible to achieve modularly in some of the semantic approaches.

Concurrency often implies non-determinism, which is not always desirable. For example, the first program above can also evaluate to a result configuration in which `s` is 0. This happens when the first spawned thread calculates the sum `s + m`, which is 0, but postpones writing it to `s` until all the subsequent 100 new threads complete their execution. Similarly, after the execution of the second program above, `x` can have any of the values 1, 10, 11, 100, 101, 110, 111 (see Exercise 82).

A language designer or semanticist may find it very useful to execute and analyze programs like above in her semantics. Indeed, the existence of certain behaviors or their lack of, may suggest changes in the syntax and the semantics of the language at an early and therefore cheap design stage. For example, one may decide that one's language must be race-free on any variable except some special semaphore variables used specifically for synchronization purposes (this particular decision may be too harsh on implementations, though, which may end up having to rely on complex static analysis front-ends or even ignoring it). We make no such difficult decisions in our simple language here, limiting our goal to the bottom of the spectrum of semantic possibilities: we only aim at giving a semantics that faithfully captures all the program behaviors due to spawning threads.

We make the simplifying assumptions that we have a sequentially consistent memory and that variable read and write operations are atomic and thus unaffected by potential races. For example, if `x` is 0 in a two-threaded program where one thread is about to write 5 to `x` and the other is about to read `x`, then the only global next steps can be either that the first thread writes 5 to `x` or that the second thread reads 0 as the value of `x`; in other words, we assume that it is impossible for the second thread to read 5 or any other value except 0 as the next small-step step in the program.

Big-Step SOS

Big-step SOS and denotational semantics are the semantical approaches which are the most affected by concurrency extensions of the base language. That is because their holistic view of computation makes it hard or impossible to capture the fine-grained execution steps and behavior interleavings that are inherent to concurrent program executions. Consider, for example, a statement of the form $\texttt{spawn}\, S_1 \,;\, S_2$ in some program state σ. The only thing we can do in big-step SOS is to evaluate S_1 and S_2 in some appropriate states and then to combine their resulting states into a result state. We do not have much room for imagination here: we either evaluate S_1 in state σ and then S_2 in the resulting state, or evaluate S_2 in state σ and then S_1 in the resulting state. The big-step SOS rule for sequential composition already implies the former case provided that $\texttt{spawn}\, S$ can evaluate to whatever state S evaluates to, which is true and needs to be considered anyway. Thus, we can formalize the above into the following two big-step SOS rules, which can be regarded as a rather desperate attempt to use big-step SOS for defining concurrency:

$$\frac{\langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle}{\langle\, \texttt{spawn}\, s, \sigma \rangle \Downarrow \langle \sigma' \rangle} \qquad (\textsc{BigStep-Spawn-Arg})$$

$$\frac{\langle s_2, \sigma \rangle \Downarrow \langle \sigma_2 \rangle \qquad \langle s_1, \sigma_2 \rangle \Downarrow \langle \sigma_1 \rangle}{\langle\, \texttt{spawn}\, s_1 \,;\, s_2, \sigma \rangle \Downarrow \langle \sigma_1 \rangle} \qquad (\textsc{BigStep-Spawn-Wait})$$

As expected, the two big-step SOS rules above capture only a limited number of the possible concurrent behaviors even for small and simple programs like the ones discussed above. One may try to change the entire big-step SOS definition of IMP to collect in result configurations all possible ways in which the corresponding fragments of program can evaluate. However, in spite of its non-modularity, there seems to be no easy way to combine, for example, the behaviors of S_1 and of S_2 into the behaviors of $\texttt{spawn}\, S_1 \,;\, S_2$.

Type System using Big-Step SOS

From a typing perspective, `spawn` is nothing but a language construct expecting a statement as argument and producing a statement as result. To type programs using `spawn` statements we therefore add the following typing rule to the already existing typing rules in Figure 3.10:

$$\frac{\mathit{xl} \vdash s : \mathit{stmt}}{\mathit{xl} \vdash \texttt{spawn}\, s : \mathit{stmt}} \qquad (\textsc{BigStepTypeSystem-Spawn})$$

Small-Step SOS

Small-step semantics are more appropriate for concurrency, because they allow a finer-grain view of computation. For example, they allow to say that the next computational step of a statement of the form $\texttt{spawn}\, S_1 \,;\, S_2$ comes either from S_1 or from S_2 (which is different from saying that either

S_1 or S_2 is evaluated next all the way through, like in big-step SOS). Since `spawn` S_1 is already permitted to advance by the small-step SOS rule for sequential composition, the following three small-step SOS rules achieve the desired behavioral non-determinism caused by concurrent threads:

$$\frac{\langle s, \sigma\rangle \to \langle s', \sigma'\rangle}{\langle\, \texttt{spawn}\, s, \sigma\rangle \to \langle\, \texttt{spawn}\, s', \sigma'\rangle} \qquad (\textsc{SmallStep-Spawn-Arg})$$

$$\langle\, \texttt{spawn skip}\,, \sigma\rangle \to \langle\, \texttt{skip}\,, \sigma\rangle \qquad (\textsc{SmallStep-Spawn-Skip})$$

$$\frac{\langle s_2, \sigma\rangle \to \langle s_2', \sigma'\rangle}{\langle\, \texttt{spawn}\, s_1 \,\texttt{;}\, s_2, \sigma\rangle \to \langle\, \texttt{spawn}\, s_1 \,\texttt{;}\, s_2', \sigma'\rangle} \qquad (\textsc{SmallStep-Spawn-Wait})$$

The rule (SmallStep-Spawn-Skip) cleans up terminated threads.

Unfortunately, the three rules above are not sufficient to capture all the intended behaviors. Consider, for example, the second program at the beginning of Section 3.5.4. That program was intended to have seven different behaviors with respect to the final value of x. Our current small-step SOS misses two of those behaviors, namely those in which x results in 1 and 100, respectively.

In order for the program to terminate with $x = 1$, it needs to start the first new thread, calculate the sum `x + 1` (which is 1), then delay writing it back to x until after the second and the main threads do their writes of x. However, in order for the main thread to be allowed to execute its assignment statement, the two grouped `spawn` statements need to either terminate and become `skip` so that the rule (SmallStep-Seq-Skip) (see Figure 3.15) applies, or to reduce to only one `spawn` statement so that the rule (SmallStep-Spawn-Wait) above applies. Indeed, these are the only two rules which allow access to the second statement in a sequential composition. The first case is not possible, because, as explained, the first newly created thread cannot be terminated. In order for the second case to happen, since the first `spawn` statement cannot terminate, the only possibility is for the second `spawn` statement to be executed all the way through (which is indeed possible, thanks to the rules (SmallStep-Seq-Arg1) in Figure3.15 and (SmallStep-Spawn-Wait) above) and then eliminated. To achieve this elimination, we may think of adding a new rule, which appears to be so natural that one may even wonder "how did we miss it in our list above?":

$$\langle\, \texttt{spawn}\, s \,\texttt{;}\ \texttt{skip}\,, \sigma\rangle \to \langle\, \texttt{spawn}\, s, \sigma\rangle$$

This rule turns out to be insufficient and, once we fix the semantics properly, it will actually become unnecessary. Nevertheless, once we add this rule, the resulting small-step SOS can also produce the behavior in which $x = 1$ at the end of the execution of the second program at the beginning of Section 3.5.4. However, it is still insufficient to produce the behavior in which $x = 100$.

In order to produce a behavior in which $x = 100$ when executing the second program, the main thread should first execute its `x + 100` step (which evaluates to 100), then let the two child threads do their writes to x, and then write the 100 to x. We have, unfortunately, no rule that allows computations within s_2 in a sequential composition s_1 ; s_2 where s_1 is different from `skip` or a `spawn` statement, as it is our case here. What we want is some generalization of the rule (SmallStep-Spawn-Wait) above which allows computations in s_2 whenever it is preceded by a sequence of spawns. On paper definitions, one can do that rather informally by means of some informal side condition saying so. If one needs to be formal, which is a must when one needs to execute the resulting language definitions as we do here, one can define a special sequent saying that a statement only spawns new threads and does nothing else (in the same spirit as the the $C\surd$

sequents in Exercise 48), and then use it to generalize the rule (SmallStep-Spawn-Wait) above. However, that would be a rather particular and potentially non-modular (what if later on we add agents or other mechanisms for concurrency?) solution.

Our general solution is to instead enforce the sequential composition of IMP to be structurally associative, using the following structural identity:

$$(s_1 \; ; \, s_2) \; ; \, s_3 \equiv s_1 \; ; \, (s_2 \; ; \, s_3) \qquad \text{(SmallStep-Seq-Assoc)}$$

That means that the small-step SOS reduction rules now apply *modulo* the associativity of sequential composition, that is, that it suffices to find a structurally equivalent representative of a syntactic term which allows a small-step SOS rule to apply. In our case, the original program is structurally equivalent to one whose first statement is the first `spawn` and whose second statement is the sequential composition of the second `spawn` and the assignment of the main thread, and that structurally equivalent program allows all seven desired behaviors, so the original program also allows them. It is important to understand that we cannot avoid enforcing associativity (or, alternatively, the more expensive solution discussed above) by simply parsing the original program so that we start with a right-associative arrangement of the sequentially composed statements. The problem is that right-associativity may be destroyed as the program executes, for example when applying the true/false rules for the `if` statement, so it needs to be dynamically enforced.

Structural identities are not easy to execute and/or implement, because they can quickly yield an exponential explosion in the number of terms that need to be matched by rules. Since in our particular case we only need the fully right-associative representative of each sequential composition, we can even replace the structural identity above by a small-step SOS rule. The problem with this is that the intended computational granularity of the language is significantly modified; for example, the application of a true/false rule for the conditional statement may trigger as many such artificial rearrangement steps as statements in the chosen branch, to an extent that such rearrangement steps could dominate the total number of steps seen in some computations.

Denotational Semantics

As already mentioned when we discussed the big-step SOP of `spawn` above, big-step SOS and denotational semantics are the semantic approaches which are the most affected by the addition of concurrency to IMP. While big-step SOS was somewhat able to capture some of the non-determinism due to concurrency, unfortunately, denotational semantics cannot do even that easily. The notes on denotational semantics in Section 3.4.3 mention the use of powerdomains and resumptions when giving concurrent semantics to languages. These are complex denotational semantics topics, which are not easy to use even by experts. Moreover, they yield semantics which are either non-executable at all or very slow. Since the main emphasis of this book is on operational semantics, we do not discuss these advanced topics in this book. Instead, we simply dissolve the `spawn` statements, so we can still execute IMP++ programs using denotational semantics:

$$[\![\texttt{spawn}\, s]\!] = [\![s]\!]$$

Of course, spawning threads is completely useless with our denotational semantics here.

3.5.5 Adding Local Variables

Blocks with local variable declarations are common to many imperative, object oriented and functional languages. In IMP++ we follow the common notation where a block is a sequence of statements surrounded with curly brackets. Variables can be declared anywhere inside a block, their *scope* being the rest of the block (whatever follows the declaration); in other words, a declared variable is not visible before its declaration or outside the block declaring it. A declaration of a variable that appears in the scope of another declaration of the same variable is said to *shadow* the original one. For example, the values of `y` and `z` are 1 and 2, respectively, right before the end of the following two IMP++ blocks (none of the variables are visible outside the given blocks):

```
{  var x,y,z ;
   x := 1 ;
   y := x ;
   var x ;
   x := 2 ;
   z := x  }
```

```
{  var x,y,z ;
   x := 1 ;
   {  var x ;
      x := 2 ;
      z := x  } ;
   y := x  }
```

As already explained in the preamble of Section 3.5, the introduction of blocks and local variables suggests some syntactic and semantic simplifications in the already existing definition of IMP. For example, since local variable declarations generalize the original global variable declarations of IMP, there is no need for the original global declarations. Thus, programs can be just statements. Therefore, we remove the top-level variable declaration and add the following new syntax:

$$\begin{array}{rcl} \mathit{Stmt} & ::= & \{\} \\ & | & \{\,\mathit{Stmt}\} \\ & | & \texttt{var}\ \mathbf{List}\{\mathit{Id}\} \\ \mathit{Pgm} & ::= & \mathit{Stmt} \end{array}$$

In each of the semantics, we assume that all the previous rules referring to global variable declarations are removed. Moreover, for semantic clarity, we assume that variable declarations can only appear in blocks (a hypothetical parser can reject those programs which do not conform).

An immediate consequence of this language extension and the conventions above is that programs now evaluate to empty states. Indeed, since the initial state in which a program is evaluated is empty and since variable declarations are local to the blocks in which they occur, the state obtained after evaluating a program is empty. This makes it somewhat difficult to test this IMP extension. To overcome this problem, one can either add an output statement to the language like in Section 3.5.2 (we will do this in Section 3.5.6), or one can manually initialize the state with some "global" variables and then use those variables undeclared in the program.

It would be quite tedious to give semantics directly to the syntactic constructs above. Instead, we are going to propose another construct which is quite common and easy to give semantics to in each of the semantic approaches, and then statically translate the constructs above into the new construct. The new construct has the following syntax:

$$\mathit{Stmt} \ ::= \ \texttt{let}\ \mathit{Id} = \mathit{AExp}\ \texttt{in}\ \mathit{Stmt}$$

Its semantics is as expected: the arithmetic expression is first evaluated to an integer, then the declared variable is bound to that integer possibly shadowing an already existing binding of the same variable, then the statement is evaluated in the new state, and finally the environment before the execution of the `let` is recovered. The latter step is executed by replacing the value of the variable after the execution of the statement with whatever it was before the `let`, possibly undefined. All the other side effects generated by the statement are kept.

Here we propose a simple set of macros which automatically desugar any program using the block and local variable constructs into a program containing only `let` and the existing IMP constructs:

$$\begin{array}{rcl}
(s_1 \mathrel{\texttt{;}} s_2) \mathrel{\texttt{;}} s_3 & = & s_1 \mathrel{\texttt{;}} (s_2 \mathrel{\texttt{;}} s_3) \\
\texttt{var}\, xl \texttt{,}\, x \mathrel{\texttt{;}} s & = & \texttt{var}\, xl \mathrel{\texttt{;}} \texttt{let}\ x \texttt{=} 0\ \texttt{in}\ s \\
\texttt{var} \cdot \mathrel{\texttt{;}} s & = & s \\
s \mathrel{\texttt{;}} \texttt{var}\, xl & = & s \\
\texttt{\{}\ \texttt{var}\, xl\ \texttt{\}} & = & \texttt{skip} \\
\texttt{\{}\ s\ \texttt{\}} & = & s \\
\texttt{\{\}} & = & \texttt{skip}
\end{array}$$

The macros above are expected to be iteratively applied in order, from top to bottom, until no macro can be applied anymore. When that happens, there will be no block or variable declaration left; all of these would have been systematically replaced by the `let` construct. The first macro enforces right-associativity of sequential composition. This way, any non-terminal variable declaration (i.e., one which is not the last statement in a block) will be followed, via a sequential composition, by the remainder of the block. The second and the third macros iteratively replace each non-terminal variable declaration by a corresponding `let` statement, while the fourth and the fifth eliminate the remaining (and useless) terminal variable declarations. Finally, the sixth and the seventh macros eliminate the blocks, which are unnecessary now. From here on we assume that these syntactic desugaring macros are applied statically, before any of the semantic rules is applied; this way, the subsequent semantics will only be concerned with giving semantics to the `let` construct.

Big-Step SOS

The big-step SOS rule of `let` follows quite closely its informal description above:

$$\frac{\langle a,\sigma\rangle \Downarrow \langle i\rangle \qquad \langle s,\sigma[i/x]\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{let}\ x \texttt{=} a\ \texttt{in}\ s,\sigma\rangle \Downarrow \langle \sigma'[\sigma(x)/x]\rangle} \qquad (\textsc{BigStep-Let})$$

In words, the arithmetic expression a is first evaluated to some integer i. Then the statement s is evaluated in state $\sigma[i/x]$, resulting in a state σ'. Then we return $\sigma'[\sigma(x)/x]$ as the result of the `let` statement, that is, the state σ' in which we update the value of x to whatever x was bound to originally in σ. We *cannot* return σ' as the result of the `let`, because σ' binds x to some value which is likely different from what x was bound to in σ (note that s is allowed to assign to x, although that is not the main problem here). If σ is undefined in x, that is, if $\sigma(x) = \bot$, then $\sigma'[\sigma(x)/x]$ is also undefined in x: indeed, recall from Section 2.3.2 that $\sigma'[\bot/x]$ "undefines" σ' in x.

Since programs are now just statements, their big-step SOS simply reduces to that of statements:

$$\frac{\langle s,\cdot\rangle \Downarrow \langle \sigma\rangle}{\langle s\rangle \Downarrow \langle \sigma\rangle} \qquad (\textsc{BigStep-Pgm})$$

Hence, programs are regarded as statements that execute in the empty state. However, since variable accesses in IMP require the variable to be declared and since all variable declarations are translated into `let` statements, which recover the state in the variable they bind after their execution, we can conclude that σ will always be empty whenever a sequent of the form $\langle s\rangle \Downarrow \langle \sigma\rangle$ is derivable using the big-step SOS above. The rule above is, therefore, not very practical. All it tells us is that if $\langle s\rangle \Downarrow \langle \sigma\rangle$ is derivable then s is a well-formed program which terminates. The idea of reducing the semantics of statement-programs to that of statements in an initial state like above is general though, and it becomes practical when we add other features to the language, for example output, as we do in IMP++ (see Section **??**).

Type System using Big-Step SOS

Following the intuitions above, to type programs using `let` statements we add the following typing rules to the already existing typing rules in Figure 3.10:

$$\frac{xl \vdash a : int \qquad xl, x \vdash s : stmt}{xl \vdash \texttt{let}\ x = a\ \texttt{in}\, s : stmt} \qquad \text{(BigStepTypeSystem-Let)}$$

$$\frac{\cdot \vdash s : stmt}{\vdash s : pgm} \qquad \text{(BigStepTypeSystem-Pgm)}$$

Small-Step SOS

The small-step SOS of `let` $x = a$ `in` s can be described in words as follows: first evaluate a stepwise, until it becomes some integer; then evaluate s stepwise in a state originally binding x to the integer to which a evaluates, but making sure that the value bound to x is properly updated during each step in the evaluation of s and it is properly recovered after each step to whatever it was in the environment outside the `let` (so other potentially interleaved rules taking place outside the `let` see a consistent state); finally, dissolve the `let` when its enclosed statement becomes `skip`. All these can be achieved with the following three rules, without having to change anything in the already existing small-step SOS of IMP:

$$\frac{\langle a, \sigma \rangle \to \langle a', \sigma \rangle}{\langle \texttt{let}\ x = a\ \texttt{in}\, s, \sigma \rangle \to \langle \texttt{let}\ x = a'\ \texttt{in}\, s, \sigma \rangle} \qquad \text{(SmallStep-Let-Exp)}$$

$$\frac{\langle s, \sigma[i/x] \rangle \to \langle s', \sigma' \rangle}{\langle \texttt{let}\ x = i\ \texttt{in}\, s, \sigma \rangle \to \langle \texttt{let}\ x = \sigma'(x)\ \texttt{in}\, s', \sigma'[\sigma(x)/x] \rangle} \qquad \text{(SmallStep-Let-Stmt)}$$

$$\langle \texttt{let}\ x = i\ \texttt{in skip}, \sigma \rangle \to \langle \texttt{skip}, \sigma \rangle \qquad \text{(SmallStep-Let-Done)}$$

Note that if x was undeclared before the `let` then so it stays after each application of the rule (SmallStep-Let-Stmt), because $\sigma'[\bot/x]$ "undefines" σ' in x (see Section 2.3.2).

Like in big-step SOS, the semantics of programs (which are now statements) reduces to that of statements. One simple way to achieve that in small-step SOS is to add a rule $\langle s \rangle \to \langle s, \cdot \rangle$, in the same spirit as the small-step SOS of IMP in Section 3.3.2. However, like in Exercise 51, one could argue that this approach is wasteful, since one does not want to spend a step only to initialize the empty state (this can be regarded as poor style). For demonstration purposes and for the sake of a semantic variation, we next show a non-wasteful small-step SOS rule of programs:

$$\frac{\langle s, \cdot \rangle \to \langle s', \sigma \rangle}{\langle s \rangle \to \langle s', \sigma \rangle} \qquad \text{(SmallStep-Pgm)}$$

One could still argue that the rule above is not perfect, because the configuration $\langle \texttt{skip} \rangle$ is frozen; thus, while any other (terminating) program eventually reduces to a configuration of the form $\langle \texttt{skip}, \cdot \rangle$, `skip` itself does not. To address this non-uniformity problem, one can add a rule $\langle \texttt{skip} \rangle \to \langle \texttt{skip}, \cdot \rangle$; this wastes a step, indeed, but this case when the entire program is just `skip` is expected to be very uncommon. A conceptually cleaner alternative is to replace the rule (SMALLSTEP-PGM) above with a structural identity $\langle s \rangle \equiv \langle s, \cdot \rangle$ identifying each program configuration with a statement configuration holding an empty state. This can be easily achieved in Maude using an equation, but it can be harder to achieve in other rewrite systems providing support and semantics only for rewrite rules but not for equations.

Denotational Semantics

The denotational semantics of the `let` construct is very compact and elegant:

$$[\![\texttt{let}\ x = a\ \texttt{in}\ s]\!]\sigma = ([\![s]\!](\sigma[[\![a]\!]\sigma/x]))[\sigma(x)/x] \qquad \text{(DENOTATIONAL-LET)}$$

Like in the other semantics above, this works because $\sigma'[\bot/x]$ "undefines" σ' in x (see Section 2.3.2).

3.5.6 Putting Them All Together

We next further analyze the modularity of the various semantic approaches by defining the IMP++ language, which puts together all the language features discussed so far. Recall from the preamble of Section 3.5 that, syntactically, IMP++ removes from IMP the global variable declarations and adds the following constructs:

$$\begin{array}{rcl}
AExp & ::= & \texttt{++}\,Id \quad | \quad \texttt{read()} \\
Stmt & ::= & \texttt{print(}\,AExp\,\texttt{)} \\
 & | & \texttt{halt} \\
 & | & \texttt{spawn}\,Stmt \\
 & | & \texttt{\{\}} \quad | \quad \texttt{\{}\,Stmt\,\texttt{\}} \quad | \quad \texttt{var}\ \textbf{List\{}Id\textbf{\}} \\
Pgm & ::= & Stmt
\end{array}$$

We consider the semantics of these constructs adopted in the previous sections.

To make the design of IMP++ more permissive in what regards its possible implementations, we shall opt for maximum non-determinism whenever such design choices can be made. For example, in the case of division expressions $a_1 \,\texttt{/}\, a_2$, we want to capture all possible behaviors (recall that division is non-deterministic) due to the possibly interleaved evaluations of a_1 and a_2, including all possible abruptly terminated behaviors generated when a_2 evaluates to 0. In particular, we want to also capture those behaviors where a_1 is not completely evaluated. The rationale for this language design decision is that we want to allow maximum flexibility to implementations of IMP++; for example, some implementations may choose to evaluate a_1 and a_2 in two concurrent threads and to stop with abrupt termination as soon as the thread evaluating a_2 yields 0.

Somewhat surprisingly, when adding several new features together to a language, it is not always sufficient to simply apply all the global, non-modular changes that are required for each feature in isolation. We sometimes have to additionally consider the semantic implications of the various combinations of features. For example, the addition of side-effects in combination with division-by-zero abrupt termination requires the addition of new rules in order to catch specific new behaviors due to this particular combination. Indeed, the evaluation of $a_1 \,\texttt{/}\, a_2$, for example, may abruptly terminate with the current state precisely when a_2 is evaluated to zero, but it can also terminate with the state obtained after evaluating a_1 or parts of a_1, as discussed above.

Also, a language design decision needs to be made in what regards the state of abruptly terminated programs. One option is to simply enclose the local state when the abrupt termination flag was issued. This option is particularly useful for debugging. However, as argued in Section 3.5.3, we want abruptly terminated programs to behave the same way as the normally terminated programs. Since normally terminated programs now empty the state after their execution, we will give the semantics of abruptly terminated programs to also empty the state. Whenever easily possible, we will give the semantics of abruptly terminated statements to return after their evaluation a state binding precisely the same variables as before their evaluation.

A design decision also needs to be made in what regards the interaction between abrupt termination and threads. We (subjectively) choose that abrupt termination applies to the entire program, no matter whether it is issued by the main program or by a spawned thread. An alternative would be that abrupt termination only applies to the spawned thread if issued by a spawned thread, or to the entire program if issued by the main program. Yet another alternative is to consider the main program as an ordinary thread, and an abrupt termination issued by the main program to only stop that thread, allowing the other spawned threads to continue their executions.

Finally, there is an interesting aspect regarding the interaction between blocks and threads. In conventional programming languages, spawned threads continue to execute concurrently with the rest of the program regardless of whether the language construct which generated them completed its execution or not. For example, if function $f()$ spawns a thread and then immediately returns 1, then the expression $f() + f()$ evaluates to 2 and the two spawned threads continue to execute concurrently with the rest of the program. We do not have functions in IMP++, but we still want the spawned threads to continue to execute concurrently with the rest of the program even after the completion of the block within which the spawn statements were executed. For example, we would like the IMP++ program (its `let`-desugared variant is shown to the right)

```
{
   var x ;
   {
      var y ;
      spawn x := x + 1
   } ;
   spawn x := x + 10 ;
   print(x)
}
```

```
let x = 0 in (
   let y = 0 in
      spawn x := x + 1 ;
   spawn x := x + 10 ;
   print(x)
   )
```

to manifest four behaviors, where x is 0, 1, 10, and 11, and not only two (where x is 1, 11) as it would be the case if the first spawn statement were not allowed to transcend its surrounding block.

Below we discuss, for each semantic approach, the changes that we have to apply to the semantics of IMP in order to extend it to IMP++, highlighting changes that cannot be mechanically derived from the changes required by each of IMP++'s features when considered in isolation.

Big-Step SOS

To accommodate the side effects generated by variable increment on the state and by `read()` on the input buffer, and the possible abrupt termination generated by division-by-zero, the arithmetic expression sequents need to change from $\langle a, \sigma \rangle \Downarrow \langle i \rangle$ to $\langle a, \sigma, \omega \rangle \Downarrow \langle i, \sigma', \omega' \rangle$ for normal termination and to $\langle a, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma', \omega' \rangle$ for abrupt termination, and similarly for boolean expressions, where σ, ω and σ',ω' are the states and input buffers before and after the evaluation of a, respectively. Also, the original elegant big-step SOS rules for expressions need to change to take into account the new configurations, the various side effects, and the abrupt termination due to division-by-zero. For example, the elegant IMP big-step SOS rule for division in Section 3.2.2, namely

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \qquad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 \; / \; a_2, \sigma \rangle \Downarrow \langle i_1 \; /_{Int} \; i_2 \rangle} \quad \text{if } i_2 \neq 0$$

changes into the following six rules:

$$\frac{\langle a_1, \sigma, \omega \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \qquad \langle a_2, \sigma_1, \omega_1 \rangle \Downarrow \langle i_2, \sigma_2, \omega_2 \rangle}{\langle a_1 \; / \; a_2, \sigma, \omega \rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_2, \omega_2 \rangle}, \quad \text{where } i_2 \neq 0$$

$$\frac{\langle a_1, \sigma, \omega \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \qquad \langle a_2, \sigma_1, \omega_1 \rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2 \rangle}{\langle a_1 \; / \; a_2, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2 \rangle}$$

$$\frac{\langle a_1, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1 \rangle}{\langle a_1 \; / \; a_2, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1 \rangle}$$

$$\frac{\langle a_1, \sigma_2, \omega_2 \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \qquad \langle a_2, \sigma, \omega \rangle \Downarrow \langle i_2, \sigma_2, \omega_2 \rangle}{\langle a_1 \; / \; a_2, \sigma, \omega \rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_1, \omega_1 \rangle}, \quad \text{where } i_2 \neq 0$$

$$\frac{\langle a_1, \sigma_2, \omega_2 \rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1 \rangle \qquad \langle a_2, \sigma, \omega \rangle \Downarrow \langle i_2, \sigma_2, \omega_2 \rangle}{\langle a_1 \; / \; a_2, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1 \rangle}$$

$$\frac{\langle a_2, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2 \rangle}{\langle a_1 \; / \; a_2, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2 \rangle}$$

Like for the individual features, rules like the above attempt to capture the intended nondeterministic evaluation strategy of the arithmetic operators, but they end up capturing only the non-deterministic choice semantics. In the case of division, we also have to add the rule for abrupt termination in the case of a division-by-zero, like the rule (BigStep-Div-By-Zero) in Section 3.5.3. However, since we want to capture all the non-deterministic behaviors that big-step SOS can detect, we actually need three such rules:

$$\frac{\langle a_1, \sigma, \omega \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \qquad \langle a_2, \sigma_1, \omega_1 \rangle \Downarrow \langle 0, \sigma_2, \omega_2 \rangle}{\langle a_1 \; / \; a_2, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2 \rangle}$$

$$\frac{\langle a_1, \sigma_2, \omega_2 \rangle \Downarrow \langle i_1, \sigma_1, \omega_1 \rangle \qquad \langle a_2, \sigma, \omega \rangle \Downarrow \langle 0, \sigma_2, \omega_2 \rangle}{\langle a_1 \; / \; a_2, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma_1, \omega_1 \rangle}$$

$$\frac{\langle a_2, \sigma, \omega \rangle \Downarrow \langle 0, \sigma_2, \omega_2 \rangle}{\langle a_1 \; / \; a_2, \sigma, \omega \rangle \Downarrow \langle \texttt{error}, \sigma_2, \omega_2 \rangle}$$

The big-step SOS sequents for statements $\langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle$ also need to change, to hold both the input/output buffers and the termination flag. We use sequents of the form $\langle s, \sigma, \omega_{in} \rangle \Downarrow \langle \sigma', \omega'_{in}, \omega_{out} \rangle$ for the case when s terminates normally and sequents of the form $\langle s, \sigma, \omega_{in} \rangle \Downarrow \langle \texttt{halting}, \sigma', \omega'_{in}, \omega_{out} \rangle$

for the case when s terminates abruptly; here $\omega_{in}, \omega'_{in} \in$ *Buffer* are the input buffers before and, respectively, after the evaluation of statement s, and $\omega_{out} \in$ *Buffer* is the output produced during the evaluation of s. All the big-step SOS rules for statements need to change to accommodate the new sequents, making sure that side effects and abrupt termination are properly propagated, like we did in Sections 3.5.1 and 3.5.3 (but for the new configurations). We also have to include big-step SOS rules for input/output like those in Section 3.5.2, for local variable declarations like those in Section 3.5.5, and for dynamic threads like those in Section 3.5.4, but also modified to work with the new configurations and to propagate abrupt termination. Recall from the preamble of this section that we want abruptly terminated programs to terminate similarly to the normal programs, that is, with an empty state in the configuration. This can be easily achieved in the rule for programs by simply emptying the state when the program statement terminates abruptly. However, in the case of big-step SOS it is relatively easy to ensure a stronger property, namely that each statement leaves the state in a consistent shape after its evaluation, no matter whether that is abruptly terminated or not. All we have to do is to also clean up the state when the halting signal is propagated through the `let` construct. For clarity, we show both big-step SOS rules for `let`:

$$\frac{\langle a, \sigma, \omega_{in}\rangle \Downarrow \langle i, \sigma', \omega'_{in}\rangle \qquad \langle s, \sigma'[i/x], \omega'_{in}\rangle \Downarrow \langle \sigma'', \omega''_{in}, \omega_{out}\rangle}{\langle \texttt{let}\, x = a \,\texttt{in}\, s, \sigma, \omega_{in}\rangle \Downarrow \langle \sigma''[\sigma'(x)/x], \omega''_{in}, \omega_{out}\rangle}$$

$$\frac{\langle a, \sigma, \omega_{in}\rangle \Downarrow \langle i, \sigma', \omega'_{in}\rangle \qquad \langle s, \sigma'[i/x], \omega'_{in}\rangle \Downarrow \langle \texttt{halting}, \sigma'', \omega''_{in}, \omega_{out}\rangle}{\langle \texttt{let}\, x = a \,\texttt{in}\, s, \sigma, \omega_{in}\rangle \Downarrow \langle \texttt{halting}, \sigma''[\sigma'(x)/x], \omega''_{in}, \omega_{out}\rangle}$$

Recall from Section 2.3.2 that $\sigma''[\bot/x]$ "undefines" σ'' in x.

Finally, the big-step SOS sequents and rules for programs also have to change, to take into account the fact that programs are now just statements like in Section 3.5.5, that they take an input and that they yield both the unconsumed input and an output like in Section 3.5.2, and that programs manifest normal termination behavior no matter whether their corresponding statement terminates normally or not:

$$\frac{\langle s, xl \mapsto 0, \omega_{in}\rangle \Downarrow \langle \sigma, \omega'_{in}, \omega_{out}\rangle}{\langle s, \omega_{in}\rangle \Downarrow \langle \sigma, \omega'_{in}, \omega_{out}\rangle}$$

$$\frac{\langle s, xl \mapsto 0, \omega_{in}\rangle \Downarrow \langle \texttt{halting}, \sigma, \omega'_{in}, \omega_{out}\rangle}{\langle s, \omega_{in}\rangle \Downarrow \langle \sigma, \omega'_{in}, \omega_{out}\rangle}$$

Unfortunately, as seen above, all the configurations, sequents and rules of IMP extended with any one of the features of IMP++ had to change again when we added all the features together. This highlights, again, the poor modularity of big-step SOS. But, even accepting the poor modularity of big-step SOS, do we at least get all the behaviors expressible in a big-step SOS style by simply putting together and adjusting accordingly all the rules of the individual features? Unfortunately, not. Recall from the preamble of Section 3.5.6 that we want spawned threads to execute concurrently with the rest of the program also after their surrounding blocks complete. In other words, we would like to also capture behaviors of, e.g., $(\texttt{let}\, x = a \,\texttt{in}\, \texttt{spawn}\, s_1)\,\texttt{;}\, s_2$, where a is first evaluated, then s_2, and then s_1. We can easily add a big-step SOS rule to capture this situation, but is that enough? Of course not, because there are many other similar patters in which we would like to allow the evaluation of s_2 before the preceding statement completes its evaluation. For example, one can replace `spawn` s_1 above by another `let` holding a spawn statement, or by `spawn` s_1' `;` `spawn` s_2', or by combinations of such patterns. A tenacious reader could probably find some complicated way to allow all these behaviors. However, it is fair to say that big-step SOS has not been conceived to deal with concurrent languages, and can only partially deal with non-determinism.

Type System using Big-Step SOS

The IMP++ type system is quite simple and modular. We simply put together all the typing rules of the individual language features discussed in Sections 3.5.1, 3.5.2, 3.5.3, 3.5.4, and 3.5.5.

Small-Step SOS

It is conceptually easy, though not entirely mechanical, to combine the ideas and changes to the original IMP small-step SOS discussed in Sections 3.5.1, 3.5.2, and 3.5.3, in order derive the small-step SOS rules of IMP++.

The arithmetic expression sequents need to change from $\langle a, \sigma\rangle \to \langle a', \sigma'\rangle$ to $\langle a, \sigma, \omega\rangle \to \langle a', \sigma', \omega'\rangle$ for normal steps and to $\langle a, \sigma, \omega\rangle \to \langle \texttt{error}, \sigma', \omega'\rangle$ for halting steps, and similarly for boolean expressions, where σ, ω and σ',ω' are the states and input buffers before and after the small-step applied to a, respectively. Also, the original small-step SOS rules for expressions need to change to take into account the new configurations, the various side effects, and the abrupt termination due to division-by-zero. For example, here are the new rules for division:

$$\frac{\langle a_1, \sigma, \omega_{in}\rangle \to \langle a_1', \sigma', \omega_{in}'\rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in}\rangle \to \langle a_1' \,/\, a_2, \sigma', \omega_{in}'\rangle}$$

$$\frac{\langle a_1, \sigma, \omega_{in}\rangle \to \langle \texttt{error}, \sigma', \omega_{in}'\rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in}\rangle \to \langle \texttt{error}, \sigma', \omega_{in}'\rangle}$$

$$\frac{\langle a_2, \sigma, \omega_{in}\rangle \to \langle a_2', \sigma', \omega_{in}'\rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in}\rangle \to \langle a_1 \,/\, a_2', \sigma', \omega_{in}'\rangle}$$

$$\frac{\langle a_2, \sigma, \omega_{in}\rangle \to \langle \texttt{error}, \sigma', \omega_{in}'\rangle}{\langle a_1 \,/\, a_2, \sigma, \omega_{in}\rangle \to \langle \texttt{error}, \sigma', \omega_{in}'\rangle}$$

$$\langle i_1 \,/\, i_2, \sigma, \omega_{in}\rangle \to \langle i_1 \,/_{\mathit{Int}}\, i_2, \sigma', \omega_{in}'\rangle \quad \text{if } i_2 \neq 0$$

$$\langle a_1 \;/\; 0, \sigma, \omega_{in}\rangle \to \langle \texttt{error}\,, \sigma, \omega_{in}\rangle$$

Note that the last rule above does not require the full evaluation of a_1 in order to flag abrupt termination. This aspect was irrelevant when we added abrupt termination in isolation to IMP in Section 3.5.3, because expressions did not have side effects there. However, since expressions can now modify both the state (via variable increment) and the input buffer (via `read()`), the rule above captures more behaviors than a rule replacing a_1 by an integer i_1, which would be obtained by mechanically translating the corresponding rule from Section 3.5.3.

The small-step SOS rules of statements and programs straightforwardly result from those of the individual features discussed in Sections 3.5.1, 3.5.2, and 3.5.3. Sequents $\langle s, \sigma\rangle \to \langle s', \sigma'\rangle$ need to change into sequents $\langle s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s', \sigma', \omega_{in}', \omega_{out}\rangle$ and $\langle s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle \texttt{halting}, \sigma', \omega_{in}', \omega_{out}\rangle$ for for normal and for halting steps, respectively, where σ, ω_{in}, ω_{out} and σ', ω'_{in}, ω'_{out} are the states, input buffers and output buffers before and after the small-step applied to s, respectively. The only rule that deviates from the expected pattern is the rule that propagates the halting signal through the `let` construct. Like in the case of big-step SOS discussed above, we can do it in such a way that the state is consistently cleaned up after each `let`, regardless of whether its body statement terminated abruptly or not. Here are all three small-step SOS rules for `let`:

$$\frac{\langle s, \sigma[i/x], \omega_{in}, \omega_{out}\rangle \to \langle s', \sigma', \omega'_{in}, \omega'_{out}\rangle}{\langle \texttt{let}\, x = i\, \texttt{in}\, s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle \texttt{let}\, x = \sigma'(x)\, \texttt{in}\, s', \sigma'[\sigma(x)/x], \omega'_{in}, \omega'_{out}\rangle}$$

$$\langle \texttt{let}\, x = i\, \texttt{in skip}, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle \texttt{skip}, \sigma, \omega_{in}, \omega_{out}\rangle$$

$$\frac{\langle s, \sigma[i/x], \omega_{in}, \omega_{out}\rangle \to \langle \texttt{halting}, \sigma', \omega'_{in}, \omega'_{out}\rangle}{\langle \texttt{let}\, x = i\, \texttt{in}\, s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle \texttt{halting}, \sigma'[\sigma(x)/x], \omega'_{in}, \omega'_{out}\rangle}$$

Recall again from Section 2.3.2 that $\sigma'[\bot/x]$ "undefines" σ' in x.

Even though strictly speaking all the small-step SOS rules above are different from their corresponding rules in the IMP extension introducing only the feature they define, we claim that they are quite mechanically derivable. In fact, MSOS (see Section 3.6) formalizes and mechanizes their derivation. The main question is then whether these new small-step SOS rules indeed capture the intended semantics of IMP++. Unfortunately, like in the case of big-step SOS, they fail to capture the intended semantics of `spawn`. Indeed, since the `let` construct does not dissolve itself until its body statement becomes `skip`, statements of the form $(\texttt{let}\, x = i\, \texttt{in spawn}\, s_1)$; s_2 will never allow reductions in s_2, thus limiting the concurrency of spawn statements to their defining blocks.

There are several ways to address the problem above, each with its advantages and limitations. One possibility is to attempt to syntactically detect all situations in which a statement allows a subsequent statement to execute, that is, all situations in which the former can only perform spawn steps. Like in Section 3.3.2 where we introduced special configurations of the form $C\surd$ (following Hennessy [36]) called "terminated configurations", we can introduce special "parallel configurations" $C||$ stating that C can only spawn statements or discard terminated spawned statements, that is, C's statement can be executed in parallel with subsequent statements. Assuming such $C||$ special configurations, we can remove the existing small-step SOS rule corresponding to rule (SmallStep-Spawn-Wait) in Section 3.5.4 and add instead the following rule:

$$\frac{\langle s_1, \sigma, \omega_{in}, \omega_{out}\rangle|| \qquad \langle s_2, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s'_2, \sigma', \omega'_{in}, \omega'_{out}\rangle}{\langle s_1 \,;\, s_2, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s_1 \,;\, s'_2, \sigma', \omega'_{in}, \omega'_{out}\rangle}$$

Alternatively, one can define $||$ as a predicate only taking a statement instead of a configuration, and then move the $||$ sequent out from the rule premise into a side condition. Nevertheless, the $||$ configurations or predicates need to be defined such that they include only statements of the form $\texttt{spawn}\, s$ and $\texttt{spawn}\, s$; $\texttt{spawn}\, s'$ and $\texttt{let}\, x = i\, \texttt{in spawn}\, s$ and combinations of such statements. The idea is that such statements can safely be executed in parallel with whatever else follows them. For example, a statement of the form $\texttt{let}\, x = a\, \texttt{in spawn}\, s$ where a is not a value does not fall into this pattern. Exercise 97 asks the reader to further explore this approach.

One problem with the approach above is that it is quite non-modular. Indeed, the definition of $||$ is strictly dependent upon the current language syntax. Adding or removing syntax to the language will require us to also revisit the definition of $||$. Another and bigger problem with this approach is that it does not seem to work for other concurrent language constructs. For example, in many languages the creation of a thread is an expression (and not a statement) construct, returning to the calling context the new thread identifier as a value. The calling context can continue its execution using the returned value in parallel with the spawned thread. In our context, for example, if $\texttt{spawn}\,s$ returned an integer value, we would have to allow expressions of the form $(\texttt{spawn}\,x := x + 1)\,\texttt{<=}\,(\texttt{spawn}\,x := x + 10) + x$ and be able to execute the two threads concurrently with the lookup for x and the evaluation of the `<=` boolean expression. It would be quite hard, if not impossible, to adapt the approach above to work with such common concurrent constructs which both return a value to the calling context and execute their code in parallel with the context.

Another way to address the loss of concurrent behaviors due to the syntactic constraints imposed by `let` on the `spawn` statements in its body, is to eliminate the `let` as soon as it is semantically unnecessary. Indeed, once the expression a is evaluated in a construct $\texttt{let}\,x = a\,\texttt{in}\,s$, the `let` is semantically unnecessary. The only reason we kept it was because it offered us an elegant syntactic means to keep track of the execution contexts both for its body statement and for the outside environment. Unfortunately, it is now precisely this "elegant syntactic means" that inhibits the intended concurrency of the `spawn` statement. One way to eliminate the semantically unnecessary `let` statements is to try to add a small-step SOS rule of the form:

$$\langle \texttt{let}\,x = i\,\texttt{in}\,s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s\ ;\ x := \sigma(x), \sigma[i/x], \omega_{in}, \omega_{out}\rangle$$

The idea here is to reduce the `let` statement to its body statement in a properly updated state, making sure that the state is recovered after the execution of the body statement. The infusion of the statement $x := \sigma(x)$ is a bit unorthodox, because $\sigma(x)$ can also be undefined; it works in our case here because we allow state updates of the form $\sigma[\bot/x]$, which have the effect to undefine σ in x (see Section 2.3.2), and the assignment rule generates such a state update. This trick is not crucial for our point here; if one does not like it, then one can split the rule above in two cases, one where $\sigma(x)$ is defined and one where is it undefined, and then add a special syntactic statement construct to undefine x in the second case. The real problem with this `let`-elimination approach is that it is simply *wrong* when we also have `spawn` statements in the language. For example, if s is a spawn statement then the `let` reduces to the `spawn` statement followed by the assignment to x; the small-step SOS rule for `spawn` allowing the spawned statement to be executed in parallel with its subsequent statement then kicks in and allows the assignment to x to be possibly evaluated before the spawned statement, thus resulting in a wrong behavior: the spawned statement should only see the x bound by its `let` construct, not the outside x (possibly undefined). A correct way to eliminate the `let` construct is to rename the bound variable into a fresh variable visible only to `let`'s body:

$$\langle \texttt{let}\,x = i\,\texttt{in}\,s, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle s[x'/x], \sigma[i/x'], \omega_{in}, \omega_{out}\rangle \qquad \text{where } x' \text{ is a fresh variable}$$

Besides having to provide and maintain (as the language changes) a substitution[3] operation and then having to pay linear or worse complexity each time a `let` is eliminated, and besides creating potentially unbounded garbage bindings in the state (e.g., the let statement can be inside a loop), the solution above appears to only solve our particular problem with our particular combination of `let` and `spawn`. It still does not seem to offer us a general solution for dealing with arbitrary constructs for concurrency, in particular with `spawn` constructs that evaluate to a

[3]See Section 4.5.3.

value which is immediately returned to the calling context, as described a few paragraphs above. For example, while it allows for renaming the variable x into a fresh variable when the expression $(\texttt{spawn}\,x := x + 1) \texttt{<=} (\texttt{spawn}\,x := x + 10) + x$ appears inside a $\texttt{let}\,x = i\;\texttt{in}\ldots$ construct, we still have no clear way to evaluate both spawn expressions and the expression containing them concurrently.

The correct solution to deal with concurrency in small-step SOS is to place all the concurrent threads or processes in a syntactic "soup" where any particular thread or process, as well as any communicating subgroups of them, can be picked and advanced one step. Since in our IMP++ language we want all threads to execute concurrently without any syntactic restrictions, we have to place all of them in some top-level "soup", making sure that each of them is unambiguously provided its correct execution environment. For example, if a thread was spawned from inside a let statement, then it should correctly see precisely the execution environment available at the place where it was spawned, possibly interacting with other threads seeing the same environment; it should not wrongly interfere with other threads happening to have had variables with the same names in their creation environments. This can be done either by using a substitution like above to guarantee that each bound variable is distinct, or by splitting the current state mapping variable identifiers to values into an *environment* mapping identifiers to memory locations and a *store* (or memory, or heap) mapping locations to values. In the latter case, each spawned thread would be packed together with its creation environment and all threads would share the store. The environment-store approach is the one that we will follow in Section 3.8 and in general in $\mathbb{K}$ in Chapter 5 and many other places in the book, so we will not insist on this approach in the context of small-step SOS. Regardless of whether one uses a substitution or an environment-store approach, to place all the threads in a top-level soup we need to devise a mechanism to propagate a newly spawned thread to the top level through each syntactic construct under which it can occur. This is tedious, but not impossible.

The morale of the discussion above is that putting features together in a language defined using small-step SOS is a highly non-trivial matter even when the language is trivial, like our IMP++. On the one hand, one has to non-modularly modify the configurations to hold all the required semantic data of all the features and then to modify all the rules to make sure that all these data is propagated and updated appropriately by each language construct. Addressing this problem is the main motivation of the MSOS approach discussed in Section 3.6. One the other hand, the desired feature interactions can require quite subtle changes to the semantics, which are sometimes hard to achieve purely syntactically. One cannot avoid the feeling that syntax is sometimes just too rigid, particularly when concurrency is concerned. This is actually one of the major motivations for the chemical abstract machine computational and semantic model discussed in Section 3.8.

Denotational Semantics

In order to accommodate all the semantic data needed by all the features, the denotation functions will now have the following types:

$$\begin{aligned}
&[\![\_]\!] : AExp \to (State \times Buffer \rightharpoonup Int \cup \{error\} \times State \times Buffer)\\
&[\![\_]\!] : BExp \to (State \times Buffer \rightharpoonup Bool \cup \{error\} \times State \times Buffer)\\
&[\![\_]\!] : Stmt \to (State \times Buffer \rightharpoonup State \times Buffer \times Buffer \times \{halting, ok\})
\end{aligned}$$

Moreover, since programs are now statements and since we want their denotation to only take an input buffer and to return a state, the remainder of the input buffer and the output buffer (recall that we deliberately discard the halting status), we replace the original denotation function of programs with the following (it is important to have a different name):

$$[\![\_]\!]_{\texttt{pgm}} : Pgm \to (Buffer \rightharpoonup State \times Buffer \times Buffer)$$

For example, the denotation of division becomes

$$[\![a_1 \,/\, a_2]\!]\pi = \begin{cases} (1^{\text{st}}(arg_1) \,/_{Int}\, 1^{\text{st}}(arg_2),\ 2^{\text{nd}}(arg_2), 3^{\text{rd}}(arg_2)) & \text{if} \quad 1^{\text{st}}(arg_1) \neq error \\ & \text{and} \quad 1^{\text{st}}(arg_2) \neq error \\ & \text{and} \quad 1^{\text{st}}(arg_2) \neq 0 \\ arg_1 & \text{if} \quad 1^{\text{st}}(arg_1) = error \\ (error, 2^{\text{nd}}(arg_2), 3^{\text{rd}}(arg_2)) & \text{if} \quad 1^{\text{st}}(arg_2) = error \\ & \text{or} \quad 1^{\text{st}}(arg_2) = 0 \end{cases}$$

where $arg_1 = [\![a_1]\!]\pi$ and $arg_2 = [\![a_2]\!](2^{\text{nd}}(arg_1), 3^{\text{rd}}(arg_1))$, the denotation of sequential composition becomes

$$[\![s_1 \,;\, s_2]\!]\pi = \begin{cases} (1^{\text{st}}(arg_2),\ 2^{\text{nd}}(arg_2),\ 3^{\text{rd}}(arg_1) : 3^{\text{rd}}(arg_2),\ 4^{\text{th}}(arg_2)) & \text{if} \quad 4^{\text{th}}(arg_1) = ok \\ arg_1 & \text{if} \quad 4^{\text{th}}(arg_1) = halting \end{cases}$$

where $arg_1 = [\![s_1]\!]\pi$ and $arg_2 = [\![s_2]\!](1^{\text{st}}(arg_1),\ 2^{\text{nd}}(arg_1))$, the denotational semantics of while loops $\texttt{while}\, b\, \texttt{do}\, s$ become the fixed-points of total functions

$$\begin{array}{rl} \mathcal{F} : & (State \times Buffer \rightharpoonup State \times Buffer \times Buffer \times \{halting, ok\}) \\ \to & (State \times Buffer \rightharpoonup State \times Buffer \times Buffer \times \{halting, ok\}) \end{array}$$

defined as

$$\mathcal{F}(\alpha)(\pi) = \begin{cases} (1^{\text{st}}(arg_3),\ 2^{nd}(arg_3),\ 3^{\text{rd}}(arg_2) : 3^{\text{rd}}(arg_3),\ 4^{\text{th}}(arg_3)) & \text{if} \quad 1^{\text{st}}(arg_1) = \texttt{true} \\ & \text{and} \quad 4^{\text{th}}(arg_2) = ok \\ arg_2 & \text{if} \quad 1^{\text{st}}(arg_1) = \texttt{true} \\ & \text{and} \quad 4^{\text{th}}(arg_2) = halting \\ (2^{\text{nd}}(arg_1),\ 3^{\text{rd}}(arg_1),\ \epsilon,\ ok) & \text{if} \quad 1^{\text{st}}(arg_1) = \texttt{false} \\ (2^{\text{nd}}(arg_1),\ 3^{\text{rd}}(arg_1),\ \epsilon,\ halting) & \text{if} \quad 1^{\text{st}}(arg_1) = error \end{cases}$$

where $arg_1 = [\![b]\!]\pi$, $arg_2 = [\![s]\!](2^{\text{nd}}(arg_1),\ 3^{\text{rd}}(arg_1))$, and $arg_3 = \alpha(1^{\text{st}}(arg_2),\ 2^{\text{nd}}(arg_2))$, and the denotation of programs is defined as the function

$$[\![s]\!]_{\texttt{pgm}}\omega = (1^{\text{st}}(arg),\ 2^{\text{nd}}(arg),\ 3^{\text{rd}}(arg))$$

where $arg = [\![s]\!](\cdot, \omega)$, that is, the program statement is evaluated in the empty state and the halting flag is discarded.

Once all the changes above are applied in order to correctly handle the semantic requirements of the various features of IMP++, the denotational semantics of those features is relatively easy, basically a simple adaptation of their individual denotational semantics in Sections 3.5.1, 3.5.2, 3.5.3, 3.5.4, and 3.5.5. We let their precise definitions as an exercise to the reader (see Exercise 100). Note, however, that the resulting denotational semantics of IMP++ is still non-deterministic and non-concurrent. Because of this accepted limitation, we do not need to worry about the loss of concurrent behaviors due to the interaction between `spawn` and `let`.

3.5.7 Notes

The first to directly or indirectly pinpoint the limitations of plain SOS and denotational semantics when defining non-trivial languages were the inventors of alternative semantic frameworks, such as Berry and Boudol [10, 11] who proposed the chemical abstract machine model (see Section 3.8), Felleisen and his collaborators [29, 99] who proposed evaluation contexts (see Section 3.7), and Mosses and his collaborators [60, 61, 62] who proposed the modular SOS approach (see Section 3.6). Among these, Mosses is perhaps the one who most vehemently criticized the lack of modularity of plain SOS, bringing as evidence natural features like the ones we proposed for IMP++, which require the structure of the configurations and implicitly the existing rules to change no matter whether there is any semantic interaction or not between the new and the old features.

The lack of modularity of SOS was visible even in Plotkin's original notes [70, 71], where he had to modify the definition of simple arithmetic expressions several times as his initial language evolved. Hennessy also makes it even more visible in his book [36]. Each time he adds a new feature, he also has to change the configurations and the entire existing semantics, similarly to each of our IMP extensions in this section. However, the lack of modularity of language definitional frameworks was not perceived as a major problem until late 1990es, partly because there were few attempts to give complete and rigorous semantics to real programming languages. Hennessy actually used each language extension as a pedagogical opportunity to teach the reader what new semantic components the feature needs and how and where those are located in each sequent. Note also that Hennessy's languages were rather simple and pure. His imperative language, called *WhileL*, was actually simpler even than our IMP (*WhileL* had no global variable declarations). Hennessy's approach was somewhat different from ours, namely he defined a series of different paradigmatic languages, each highlighting certain semantic aspects in a pure form, without including features that lead to complex semantic interactions (like the side effects, blocks with local variables, and threads as in our IMP++).

Wadler [97] proposes a language design experiment similar in spirit to our extensions of IMP in this section. His core language is purely functional, but some of its added features overlap with those of IMP and IMP++: state and side effects, output, non-determinism. Wadler's objective in [97] was to emphasize the elegance and usefulness of monads in implementing interpreters in pure functional languages like Haskell, but his motivations for doing so are similar to ours: the existing (implementation or semantic) approaches for programming language design are not modular enough. Monads can also be used to add modularity to denotational semantics, to avoid having to modify the mathematical domains into products of domains as we did in this section. The monadic approach to denotational semantics will be further discussed in Section 3.9. However, as also discussed in Section 3.4.3, the denotational approach to non-determinism and concurrency is to collect all possible behaviors, hereby programs evaluating to sets or lists of values. The same holds true in Wadler's monadic approach to implement interpreters in [97]. The problem with this is that the resulting interpreters or executable semantics are quite inefficient. Contrast that with the small-step SOS approach in Section 3.3 which allows us, for example using our implementation of it in Maude, to both execute programs non-deterministically, making a possibly non-deterministic choice at each step, and search for all possible behaviors of programs.

3.5.8 Exercises

Variable Increment

The exercises below refer to the IMP extension with variable increment discussed in Section 3.5.1.

Exercise 66. *Add variable increment to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*

2. *Translate the proof system at 1 above into a rewriting logic theory, like in Figure 3.8;*

3. ☆ *Implement in Maude the rewriting logic theory at 2 above, like in Figure 3.9. To test it, add to the* IMP *programs in Figure 3.4 the following two:*

```
op sum++Pgm : -> Pgm .
eq sum++Pgm = (
     var m, n, s ;
     n := 100 ;
     while (++ m <= n) do (s := s + m)
) .
```

```
op nondet++Pgm : -> Pgm .
eq nondet++Pgm = (
     var x ;
     x := 1 ;
     x := ++ x / (++ x / x)
) .
```

The first program should have only one behavior, so, for example, both Maude commands below

```
rewrite < sum++Pgm > .
search  < sum++Pgm > =>! Cfg:Configuration .
```

should show the same result configuration, `< m |-> 101 & n |-> 100 & s |-> 5050 >`*. The second program should have (only) three different behaviors under big-step semantics; the first command below will show one of the three behaviors, but the second will show all three of them:*

```
rewrite < nondet++Pgm > .
search  < nondet++Pgm > =>! Cfg:Configuration .
```

The three behaviors captured by the big-step SOS discussed in Section 3.5.1 result in configurations `< x |-> 1 >`*,* `< x |-> 2 >`*, and* `< x |-> 3 >`*. As explained in Section 3.5.1, this big-step SOS should not be not able to expose the behaviors in which* `x` *is* `0` *and in which a division by zero takes place.*

Exercise 67. *Type* IMP *extended with variable increment:*

1. *Translate the big-step rule above into a rewriting logic rule that can be added to those in Figure 3.11 corresponding to the type system of* IMP*;*

2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the two additional programs in Example 66.*

Exercise 68. *Same as Exercise 66, but for small-step SOS instead of big-step SOS.*
☆ *Make sure that the small-step definition in Maude exhibits all five behaviors of program* `nondet++Pgm` *defined in Exercise 66 (the three behaviors exposed by the big-step definition in Maude in Exercise 66, plus one where* `x` *is* `0` *and one where the program gets stuck right before a division by zero).*

Exercise 69. *Same as Exercise 66, but for denotational semantics instead of big-step SOS.*
☆ *The definition in Maude should lack any non-determinism, so only one behavior should be observed for any program, including* `nondet++Pgm` *in Exercise 66.*

Input/Output

The exercises below refer to the IMP extension with input/output discussed in Section 3.5.2.

Exercise 70. *Add input/output to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*

2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*

3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, add to the* IMP *programs in Figure 3.4 the following three:*

```
op sumIOPgm : -> Pgm .
eq sumIOPgm = (
     var m, n, s ;
     n := read() ;
     while (m <= n) do (
       print(m) ;
       s := s + m ;
       m := m + 1
     ) ;
     print(s)
) .
```

```
op whileIOPgm : -> Pgm .
eq whileIOPgm = (
     var s ;
     s := 0 ;
     while not(read() <= 0) do
       s := s + read() ;
     print(s)
) .
```

```
op nondetIOStmt : -> Stmt .
eq nondetIOStmt = (
     print(read()
           / (read()
              / read()))
) .
```

The first two programs are deterministic, so both the rewrite and the search commands should only show one solution. The initial configuration in which the first program is executed should contain at least one integer in the input buffer, otherwise it does not evaluate; for example, the initial configuration `< sumIOPgm, 100 >` *yields a result configuration whose input buffer is empty and whose output buffer contains the numbers 0,1,2,...,100,5050. The initial configuration in which the second program is executed should eventually contain some 0 on an odd position in the input buffer; for example,* `< whileIOPgm,10:1:17:2:21:3:0:5:8:-2:-5:10 >` *yields a result configuration whose input buffer still contains the remaining input* `5:8:-2:-5:10` *and whose output buffer contains only the integer* `6`*. The third program, which is actually a statement, is non-deterministic. Unfortunately, big-step SOS misses behaviors because, as explained, it only achieves a non-deterministic choice semantics. For example, the commands*

```
rewrite < nondetIOStmt, .State, 10 : 20 : 30 > .
search  < nondetIOStmt, .State, 10 : 20 : 30 > =>! Cfg:Configuration .
```

yield configurations `< .State,epsilon,10 >` *and, respectively,* `< .State,epsilon,10 >` *and* `< .State,epsilon,15 >`*. The configuration whose output is 6 (i.e.,* $20/_{\mathit{Int}}(30/_{\mathit{Int}}10)$*) and the three undefined configurations due to division by zero are not detected.*

Exercise 71. *Type* IMP *extended with input/output:*

1. *Translate the discussed big-step SOS typing rules for input/output into corresponding rewriting logic rules that can be added to the already existing rewrite theory in Figure 3.11 corresponding to the type system of* IMP*;*

2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the additional programs in Example 70.*

Exercise 72. *Same as Exercise 70, but for small-step SOS instead of big-step SOS. Make sure that the resulting small-step SOS definition detects all six behaviors of* `nondetIOStmt` *when executed with the input buffer* `10:20:30`

Exercise 73. *Same as Exercise 70, but for denotational semantics instead of big-step SOS. Since our denotational semantics is not nondeterministic, only one behavior of* `nondetIOStmt` *is detected. Interestingly, since our denotational semantics of division was chosen to evaluate the two expressions in order, it turns out that the detected behavior is undefined (due to a division by zero). Note that although it also misses non-deterministic behaviors, big-step SOS can still detect (and even search for) valid behaviors of non-deterministic programs (see Exercise 70, where it generated a valid behavior by rewriting and found one additional behavior by searching).*

Abrupt Termination

The exercises below refer to the IMP extension with abrupt termination discussed in Section 3.5.3.

Exercise 74. *Add abrupt termination to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*
2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*
3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, execute the two programs at the beginning of Section 3.5.3. The resulting big-step SOS definition may be very slow when executed in Maude, even for small values of* `n` *(such as 2,3,4 instead of 100), which is normal (the search space is now much larger).*

Exercise 75. *Same as Exercise 74, but using a specific* `top` *construct as explained in Section 3.5.3 to catch the halting signal instead of the existing* `var` *which has a different purpose in the language.*

Exercise 76. *Type* IMP *extended with abrupt termination:*

1. *Translate the big-step SOS typing rule of* `halt` *into a corresponding rewriting logic rule that can be added to the already existing rewrite logic theory in Figure 3.11;*
2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the additional programs in Example 74.*

Exercise 77. *Same as Exercise 74, but for the small-step SOS instead of big-step SOS.*

Exercise 78. *Same as Exercise 74, but use the* `top` *construct approach instead of the rule* (SmallStep-Halting)*, to avoid wasting one computational step.*

Exercise 79. *One could argue that the introduction of the halting configurations* $\langle \texttt{halting}, \sigma\rangle$ *was unnecessary, because we could have instead used the already existing configurations of the form* $\langle \texttt{halt}, \sigma\rangle$*. Give an alternative small-step SOS definition of abrupt termination which does not add special halting configurations. Can we avoid the introduction of the* `top` *construct discussed above? Comment on the disadvantages of this approach.*

Exercise 80. *Same as Exercise 74, but for denotational semantics instead of big-step SOS.*

Exercise 81. *Same as Exercise 80, but modifying the denotation of assignment so that it is always undefined when the assigned variable has not been declared.*

Dynamic Threads

The exercises below refer to the IMP extension with dynamic threads discussed in Section 3.5.4.

Exercise 82. *Consider the two programs at the beginning of Section 3.5.4. Propose hypothetical executions of the second program corresponding to any of the seven possible values of* `x`*. What is the maximum value that* `s` *can have when the first program, as well as all its dynamically created threads, terminate? Is there some execution of the first program corresponding to each smaller value of* `s` *(but larger than or equal to 0)?*

Exercise 83. *Add dynamic threads to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*

2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*

3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, execute the two programs at the beginning of Section 3.5.4. The resulting Maude big-step SOS definition may be slow on the first program for large initial values for n: even though it does not capture all possible behaviors, it still comprises many of them. For example, searching for all the result configurations when $n = 10$ gives 12 possible values for s, namely 55,56,...,66. On the other hand, the second program only shows one out of the seven behaviors, namely the one where x results in 111.*

Exercise 84. *Same as Exercise 83, but for the type system instead of big-step SOS.*

Exercise 85. *Same as Exercise 83, but for small-step SOS instead of big-step SOS. When representing the resulting small-step SOS into rewriting logic, the structural identity can be expressed as a rewriting logic equation, this way capturing faithfully the intended computational granularity of the small-step SOS.* ☆ *When implementing the resulting rewriting logic theory into Maude, this equation can either be added as a normal equation (using* `eq`*) or as an* `assoc` *attribute to the sequential composition construct. The former will only be applied from left-to-right when executed using Maude rewriting and search commands, but that is sufficient in our particular case here.*

Exercise 86. *Same as Exercise 83, but for denotational semantics instead of big-step SOS.*

Local Variables

The exercises below refer to the IMP extension with blocks and local variables discussed in Section 3.5.5.

Exercise 87. ☆ *Implement in Maude the seven macros in the preamble of Section 3.5.5, which desugar blocks with local variable declarations into* `let` *constructs.*
Hint: In Maude, equations are applied in order. One should typically not rely on that, but in this case it may give us a simpler and more compact implementation.

Exercise 88. *Add blocks with local variables to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*

2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*

3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, modify the* IMP *programs in Figure 3.4 to use local variables and also add the two programs at the beginning of Section 3.5.5. To check whether the programs evaluate as expected, you can let some relevant variables purposely undeclared and bind them manually (to 0) in the initial state. When the programs terminate, you will see the new values of those variables. If you execute closed programs (i.e., programs declaring all the variables they use) then the resulting states will be empty, because our semantics of* `let` *recovers the state, so it will be difficult or impossible to know whether they executed correctly.*

Exercise 89. *Same as Exercise 88, but for the type system instead of big-step SOS.*

Exercise 90. *Same as Exercise 88, but for small-step SOS instead of big-step SOS.*

Exercise 91. *Same as Exercise 88, but for denotational semantics instead of big-step SOS.*

Putting Them All Together

The exercises below refer to the IMP extension with blocks and local variables discussed in Section 3.5.6.

Exercise 92. *Define* IMP++ *using big-step SOS, assuming that abrupt termination applies to the entire program, no matter whether the abrupt termination signal has been issued from inside a spawned thread or from the main program, and assuming that nothing special is done to enhance the parallelism of* `spawn` *within* `let` *blocks:*

1. *Write the complete big-step SOS as a proof system;*

2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*

3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, propose five tricky* IMP++ *programs. You programs will be added to our test-suite before grading this exercise. You get extra-points if your programs reveal limitations in the number of behaviors captured by other students' definitions.*

Exercise 93. *Same as Exercise 92, but assume that abrupt termination applies to the issuing thread only, letting the rest of the threads continue. The main program is considered to be a thread itself.*

Exercise 94. *Same as Exercise 92, but for the type system instead of big-step SOS.*

Exercise 95. *Same as Exercise 92, but for small-step SOS instead of big-step SOS.*

Exercise 96. *Same as Exercise 93, but for small-step SOS instead of big-step SOS.*

Exercise* 97. *Same as Exercise 95, but define and use "parallel configurations" of the form $C||$ in order to enhance the parallelism of* `spawn` *statements from inside* `let` *blocks.*

Exercise* 98. *Same as Exercise 97, but eliminate the* `let` *construct when semantically unnecessary using a substitution operation (which needs to also be defined).*

Exercise* 99. *Same as Exercise 98, but use an environment-store approach to the state instead of substitution.*

Exercise 100. *Same as Exercise 92, but for denotational semantics instead of big-step SOS.*