3.2 Big-Step Structural Operational Semantics (Big-Step SOS)

Known also under the names *natural semantics*, *relational semantics*, and *evaluation semantics*, big-step structural operational semantics, or *big-step SOS* for short, is the most "denotational" of the operational semantics. One can view big-step definitions as definitions of functions, or more generally of relations, interpreting each language construct in an appropriate domain. Big-step semantics is so natural, that one is strongly encouraged to use it whenever possible. Unfortunately, as discussed in Section 3.10, big-step semantics has a series of limitations making it inconvenient or impossible to use in many situations, such as when defining control-intensive features or concurrency.

A big-step SOS of a programming language or calculus is given as a formal *proof system* (see Section 2.2). The *big-step SOS sequents* are relations over configurations, typically written $C \Rightarrow R$ or $C \Downarrow R$, with the meaning that R is the configuration obtained after the (complete) evaluation of C. In this book we prefer the notation $C \Downarrow R$. A *big-step SOS rule* therefore has the form

$$\frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2 \qquad \ldots \qquad C_n \Downarrow R_n}{C \Downarrow R} \text{ [if } \textit{condition}\text{]}$$

where C, C_1, C_2, ..., C_n are configurations holding fragments of program together with all the needed semantic components, where R, R_1, R_2, ..., R_n are *result configurations*, or *irreducible configurations*, i.e., configurations which cannot be advanced anymore, and where *condition* is an optional *side condition*; as discussed in Section 2.2, the role of side conditions is to filter out undesirable instances of the rule. A big-step semantics compositionally describes how final evaluation results of language constructs can be obtained by combining the evaluation results of their syntactic counterparts (subexpressions, etc.). For example, the big-step semantics of IMP addition is

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i_1 +_{Int} i_2\rangle}$$

Here, the meaning of a relation $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ is that arithmetic expression a is evaluated in state σ to integer i. If expression evaluation has side-effects, then one has to also include a state in the right configurations, so they become of the form $\langle i, \sigma\rangle$ instead of $\langle i\rangle$, as discussed in Section 3.10.

It is common in big-step semantics to not wrap single values in configurations, that is, to write $\langle a, \sigma\rangle \Downarrow i$ instead of $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ and similarly for all the other sequents. Also, while the angle-bracket-and-comma notation $\langle \textit{code}, \textit{state}, \ldots\rangle$ is common for configurations, it is not enforced; some prefer to use a square or curly bracket notation of the form $[\textit{code}, \textit{state}, \ldots]$ or $\{\textit{code}, \textit{state}, \ldots\}$, or the simple tuple notation $(\textit{code}, \textit{state}, \ldots)$, or even to use a different (from comma) symbol to separate the various configuration ingredients, e.g., $\langle \textit{code} \mid \textit{state} \mid \ldots\rangle$, etc. Moreover, we may even encounter in the literature sequent notations of the form $\sigma \vdash a \Rightarrow i$ instead of $\langle a, \sigma\rangle \Downarrow \langle i\rangle$, as well as variants of sequent notations that prefer to move various semantic components from the configurations into special, sometimes rather informal, decorations of the symbols $\Downarrow$, $\vdash$ and/or $\Rightarrow$.

For the sake of a uniform notation, in particular when transitioning from languages whose expressions have no side effects to languages whose expressions do have side effects (as we do in Section 3.10), we prefer to always write big-step sequents as $C \Downarrow R$, and always use the angle brackets to surround both configurations involved. This solution is the most general; for example, any additional semantic data or labels that one may need in a big-step definition can be uniformly included as additional components in the configurations (the left ones, or the right ones, or both).

sorts:

Configuration

operations:

$$\begin{array}{rcl} \langle \_, \_ \rangle & : & AExp \times State \to Configuration \\ \langle \_ \rangle & : & Int \to Configuration \\ \langle \_, \_ \rangle & : & BExp \times State \to Configuration \\ \langle \_ \rangle & : & Bool \to Configuration \\ \langle \_, \_ \rangle & : & Stmt \times State \to Configuration \\ \langle \_ \rangle & : & State \to Configuration \\ \langle \_ \rangle & : & Pgm \to Configuration \end{array}$$

Figure 3.6: IMP big-step configurations as an algebraic signature.

3.2.1 IMP Configurations for Big-Step SOS

For the big-step semantics of the simple language IMP, we only need very simple configurations. We follow the comma-and-angle-bracket notational convention, that is, we separate the configuration components by commas and then enclose the entire list with angle brackets. For example, $\langle a, \sigma \rangle$ is a configuration containing an arithmetic expression a and a state σ, and $\langle b, \sigma \rangle$ is a configuration containing a Boolean expression b and a state σ. Some configurations may not need a state while others may not need the code. For example, $\langle i \rangle$ is a configuration holding only the integer number i that can be obtained as a result of evaluating an arithmetic expression, while $\langle \sigma \rangle$ is a configuration holding only one state σ that can be obtained after evaluating a statement. Configurations can therefore be of different types and need not necessarily have the same number of components. Here are all the configuration types needed for the big-step semantics of IMP:

- $\langle a, \sigma \rangle$ grouping arithmetic expressions a and states σ;
- $\langle i \rangle$ holding integers i;
- $\langle b, \sigma \rangle$ grouping Boolean expressions b and states σ;
- $\langle t \rangle$ holding truth values $t \in \{true, false\}$;
- $\langle s, \sigma \rangle$ grouping statements s and states σ;
- $\langle \sigma \rangle$ holding states σ;
- $\langle p \rangle$ holding programs p.

IMP Big-Step SOS Configurations as an Algebraic Signature

The configurations above were defined rather informally as tuples of syntax and/or states. There are many ways to rigorously formalize them, all building upon some formal definition of state (besides IMP syntax). Since we have already defined states as partial finite-domain functions (Section 3.1.2) and have already shown how partial finite-domain functions can be formalized as algebraic specifications (Section 2.3.2), we also formalize configurations algebraically.

Figure 3.6 shows an algebraic signature defining the IMP configurations needed for the subsequent big-step operational semantics. For simplicity, we preferred to explicitly define each type of needed

configuration. Consequently, our configurations definition in Figure 3.6 may be more verbose than an alternative polymorphic definition, but we believe that it is clearer for this simple language. We assumed that the sorts *AExp*, *BExp*, *Stmt*, *Pgm* and *State* come from algebraic definitions of the IMP syntax and state, like those in Sections 3.1.1 and 3.1.2; recall that the latter adapted the algebraic definition of partial functions in Section 2.3.2 (see Figure 2.1) as explained in Section 3.1.2.

3.2.2 The Big-Step SOS Rules of IMP

Figure 3.7 shows all the rules in our IMP big-step operational semantics proof system. Recall that the role of a proof system is to prove, or derive, facts. The facts that our proof system will derive have the forms $\langle a, \sigma\rangle \Downarrow \langle i\rangle$, $\langle b, \sigma\rangle \Downarrow \langle t\rangle$, $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$, and $\langle p\rangle \Downarrow \langle \sigma\rangle$ where a ranges over *AExp*, b over *BExp*, s over *Stmt*, p over *Pgm*, i over *Int*, t over *Bool*, and σ and σ' over *State*.

Informally[1], the meaning of derived triples of the form $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ is that the arithmetic expression a evaluates/executes/transitions to the integer i in state σ; the meaning of $\langle b, \sigma\rangle \Downarrow \langle t\rangle$ is similar but with Boolean values instead of integers. The reason for which it suffices to derive such simple facts is because the evaluation of expressions in our simple IMP language is side-effect-free. When we add the increment operation "`++ x`" in Section 3.10, we will have to change the big-step semantics to work with 4-tuples of the form $\langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t, \sigma'\rangle$ instead. The meaning of $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ is that the statement s takes state σ to state σ'. Finally, the meaning of pairs $\langle p\rangle \Downarrow \langle \sigma\rangle$ is that the program p yields state σ when executed in the initial state.

In the case of our simple IMP language, the transition relation is going to be *deterministic*, in the sense that $i_1 = i_2$ whenever $\langle a, \sigma\rangle \Downarrow \langle i_1\rangle$ and $\langle a, \sigma\rangle \Downarrow \langle i_2\rangle$ can be deduced (and similarly for Boolean expressions, statements, and programs). However, in the context of non-deterministic languages, triples $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ state that *a may possibly* evaluate to i in state σ, but it may also evaluate to other integers (and similarly for Boolean expressions, statements, and programs).

The proof system in Figure 3.7 contains one or two rules for each language construct, capturing its intended evaluation relation. Recall from Section 2.2 that proof rules are in fact *rule schemas*, that is, they correspond to (recursively enumerable) sets of *rule instances*, one for each concrete instance of the rule *parameters* (i.e., a, b, σ, etc.). We next discuss each of the rules in Figure 3.7.

The rules (BigStep-Lookup) and (BigStep-Int) define the obvious semantics of variable lookup and integers; these rules have no premises because variables and integers are atomic expressions, so one does not need to evaluate any other subexpression in order to evaluate them. The rule (BigStep-Add) has already been discussed at the beginning of Section 3.2, and (BigStep-Div) is similar. Note that the rules (BigStep-Lookup) and (BigStep-Div) have side conditions. We chose not to short-circuit the division operation when a_1 evaluates to 0. Consequently, no matter whether a_1 evaluates to 0 or not, a_2 is still expected to produce a correct value in order for the rule BigStep-Div to be applicable (e.g., a_2 cannot perform a division by 0).

Before we continue with the remaining rules, let us clarify, using concrete examples, what it means for rule schemas to admit multiple instances and how these can be used to derive proofs. For example, a possible instance of rule (BigStep-Div) can be the following (assume that $x, y \in Id$):

$$\frac{\langle x, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 8\rangle \qquad \langle 2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 2\rangle}{\langle x/2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 4\rangle}$$

[1]Formal definitions of these concepts can only be given after one has a formal language definition. We formally define the notions of evaluation and termination in the context of the IMP language in Definition 16.

$$\langle i, \sigma\rangle \Downarrow \langle i\rangle \qquad \text{(BigStep-Int)}$$

$$\langle x, \sigma\rangle \Downarrow \langle \sigma(x)\rangle \quad \text{if } \sigma(x) \neq \bot \qquad \text{(BigStep-Lookup)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ + } a_2, \sigma\rangle \Downarrow \langle i_1 \ +_{Int} \ i_2\rangle} \qquad \text{(BigStep-Add)}$$

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

$$\langle t, \sigma\rangle \Downarrow \langle t\rangle \qquad \text{(BigStep-Bool)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ <= } a_2, \sigma\rangle \Downarrow \langle i_1 \ \leq_{Int} \ i_2\rangle} \qquad \text{(BigStep-Leq)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle}{\langle \texttt{not } b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle} \qquad \text{(BigStep-Not-True)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle \texttt{not } b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle} \qquad \text{(BigStep-Not-False)}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \Downarrow \langle \texttt{false}\rangle} \qquad \text{(BigStep-And-False)}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \texttt{true}\rangle \qquad \langle b_2, \sigma\rangle \Downarrow \langle t\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \Downarrow \langle t\rangle} \qquad \text{(BigStep-And-True)}$$

$$\langle \texttt{skip}, \sigma\rangle \Downarrow \langle \sigma\rangle \qquad \text{(BigStep-Skip)}$$

$$\frac{\langle a, \sigma\rangle \Downarrow \langle i\rangle}{\langle x \texttt{ := } a, \sigma\rangle \Downarrow \langle \sigma[i/x]\rangle} \quad \text{if } \sigma(x) \neq \bot \qquad \text{(BigStep-Asgn)}$$

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

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle \qquad \langle s_1, \sigma\rangle \Downarrow \langle \sigma_1\rangle}{\langle \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \Downarrow \langle \sigma_1\rangle} \qquad \text{(BigStep-If-True)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle \qquad \langle s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle}{\langle \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle} \qquad \text{(BigStep-If-False)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle \texttt{while } b \texttt{ do } s, \sigma\rangle \Downarrow \langle \sigma\rangle} \qquad \text{(BigStep-While-False)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle \qquad \langle s \texttt{;while } b \texttt{ do } s, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{while } b \texttt{ do } s, \sigma\rangle \Downarrow \langle \sigma'\rangle} \qquad \text{(BigStep-While-True)}$$

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

Figure 3.7: BigStep(IMP) — Big-step SOS of IMP ($i, i_1, i_2 \in \mathit{Int}$; $x \in \mathit{Id}$; $xl \in \mathbf{List}\{\mathit{Id}\}$; $a, a_1, a_2 \in \mathit{AExp}$; $t \in \mathit{Bool}$; $b, b_1, b_2 \in \mathit{BExp}$; $s, s_1, s_2 \in \mathit{Stmt}$; $\sigma, \sigma', \sigma_1, \sigma_2 \in \mathit{State}$).

Another instance of rule (BigStep-Div) is the following, which, of course, seems problematic:

$$\frac{\langle x, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 8\rangle \qquad \langle 2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 4\rangle}{\langle x/2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 2\rangle}$$

The rule above is indeed a correct instance of (BigStep-Div), but, however, one will never be able to infer $\langle 2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 4\rangle$, so this rule can never be applied in a correct inference.

Note, however, that the following is *not* an instance of (BigStep-Div), no matter what ? is chosen to be ($\bot$, or $8/_{Int}0$, etc.):

$$\frac{\langle x, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 8\rangle \qquad \langle y, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 0\rangle}{\langle x/y, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle ?\rangle}$$

Indeed, the above does not satisfy the side condition of (BigStep-Div).

The following is a valid proof derivation, where $x, y \in Id$ and $\sigma \in State$ with $\sigma(x) = 8$ and $\sigma(y) = 0$:

$$\dfrac{\dfrac{}{\langle x, \sigma\rangle \Downarrow \langle 8\rangle} \qquad \dfrac{\dfrac{\dfrac{}{\langle y, \sigma\rangle \Downarrow \langle 0\rangle} \qquad \dfrac{}{\langle x, \sigma\rangle \Downarrow \langle 8\rangle}}{\langle y/x, \sigma\rangle \Downarrow \langle 0\rangle} \qquad \dfrac{}{\langle 2, \sigma\rangle \Downarrow \langle 2\rangle}}{\langle y/x + 2, \sigma\rangle \Downarrow \langle 2\rangle}}{\langle x/(y/x + 2), \sigma\rangle \Downarrow \langle 4\rangle}$$

The proof above can be regarded as a tree, with dots as leaves and instances of rule schemas as nodes. We call such complete (in the sense that their leaves are all dots and their nodes are correct rule instances) trees *proof trees*. This way, we have a way to mathematically *derive facts*, or *sequents*, about programs directly within their semantics. We may call the root of a proof tree the *fact (or sequent) that was proved or derived*, and the tree *its proof or derivation*.

Recall that our original intention was, for demonstration purposes, to attach various evaluation strategies to the arithmetic operations. We wanted `+` and `/` to be non-deterministic and `<=` to be left-right sequential; a non-deterministic evaluation strategy means that the subexpressions are evaluated in any order, possibly interleaving their evaluation steps, which is different from non-deterministically picking an order and then evaluating the subexpressions sequentially in that order. As an analogy, the former corresponds to evaluating the subexpressions concurrently on a multithreaded machine, while the latter to non-deterministically queuing the subexpressions and then evaluating them one by one on a sequential machine. The former has obviously potentially many more possible behaviors than the latter. Note that many programming languages opt for non-deterministic evaluation strategies for their expression constructs precisely to allow compilers to evaluate them in any order or even concurrently; some language manuals explicitly warn the reader not to rely on any evaluation strategy of arithmetic constructs when writing programs.

Unfortunately, big-step semantics is not appropriate for defining non-deterministic evaluation strategies, because such strategies are, by their nature, small-step. One way to do it is to work with sets of values instead of with values and thus associate to each fragment of program in a state the set of all the values that it can non-deterministically evaluate to. However, such an approach would significantly complicate the big-step definition, so we prefer not to do it. Moreover, since IMP has no side effects (until Section 3.10), the non-deterministic evaluation strategies would not lead to non-deterministic results anyway.

We next discuss the big-step rules for Boolean expressions. The rule (BigStep-Bool) is similar to rule (BigStep-Int), but it has only two instances, one for $t = \texttt{true}$ and one for $t = \texttt{false}$. The rules (BigStep-Not-True) and (BigStep-Not-False) are clear; they could have been combined into only one rule if we had assumed our builtin *Bool* equipped with a negation operation. Unlike the division, the conjunction has a short-circuited semantics: if the first conjunct evaluates to `false` then the entire conjunction evaluates to `false` (rule (BigStep-And-False)), and if the first conjunct evaluates to `true` then the conjunction evaluates to whatever truth value the second conjunct evaluates (rule (BigStep-And-True)).

The role of statements in a language is to change the program state. Consequently, the rules for statements derive triples of the form $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ with the meaning that if statement s is executed in state σ and *terminates*, then the resulting state is σ'. We will shortly discuss the aspect of termination in more detail. Rule (BigStep-Skip) states that `skip` does nothing with the state. (BigStep-Asgn) shows how the state σ gets updated by an assignment statement $x := a$ after a is evaluated in state σ using the rules for arithmetic expressions discussed above. (BigStep-Seq) shows how the state updates are propagated by the sequential composition of statements, and rules (BigStep-If-True) and (BigStep-If-False) show how the conditional first evaluates its condition and then, depending upon the truth value of that, it either evaluates its "then" branch or its "else" branch, but never both. The rules giving the big-step semantics of the while loop say that if the condition evaluates to `false` then the while loop dissolves and the state stays unchanged, and if the condition evaluates to `true` then the body followed by the very same while loop is evaluated (rule (BigStep-While-True)). Finally, (BigStep-Var) gives the semantics of programs as the semantics of their statement in a state instantiating all the declared variables to 0.

On Proof Derivations, Evaluation, and Termination

So far we have used the words "evaluation" and "termination" informally. In fact, without a formal definition of a programming language, there is no other way, but informal, to define these notions. Once one has a formal definition of a language, one can not only formally define important concepts like evaluation and termination, but can also rigorously reason about programs. We postpone the subject of program verification until Chapter 14; here we only define and discuss the other concepts.

Definition 16. *Given appropriate* IMP *configurations C and R, the* IMP *big-step sequent $C \Downarrow R$ is **derivable**, written* BigStep(IMP) $\vdash C \Downarrow R$*, iff there is some proof tree rooted in $C \Downarrow R$ which is derivable using the proof system* BigStep(IMP) *in Figure 3.7. Arithmetic (resp. Boolean) expression $a \in AExp$ (resp. $b \in BExp$) **evaluates** to integer $i \in Int$ (resp. to truth value $t \in \{\texttt{true}, \texttt{false}\}$) in state $\sigma \in State$ iff* BigStep(IMP) $\vdash \langle a, \sigma\rangle \Downarrow \langle i\rangle$ *(resp.* BigStep(IMP) $\vdash \langle b, \sigma\rangle \Downarrow \langle t\rangle$*). Statement s **terminates** in state σ iff* BigStep(IMP) $\vdash \langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ *for some $\sigma' \in State$; if that is the case, then we say that s **evaluates** in state σ to state σ', or that it **takes** state σ to state σ'. Finally, program p **terminates** iff* BigStep(IMP) $\vdash \langle p\rangle \Downarrow \langle \sigma\rangle$ *for some $\sigma \in State$.*

There are two reasons for which an IMP statement s may not terminate in a state σ: because it may contain a loop that does not terminate, or because it performs a division by zero and thus the rule (BigStep-Div) cannot apply. In the former case, the process of proof search does not terminate, while in the second case the process of proof search terminates in principle, but with a failure to find a proof. Unfortunately, big-step semantics cannot make any distinction between the two reasons for which a proof derivation cannot be found. Hence, the termination notion in Definition 16 rather means *termination with no error*. To catch division-by-zero within the

semantics, we need to add a special *error* value and propagate it through all the language constructs (see Exercise 41).

A formal definition of a language allows to also formally define what it means for the language to be deterministic and to also prove it. For example, we can prove that if an IMP program p terminates then there is a unique state σ such that $\textsc{BigStep}(\text{IMP}) \vdash \langle p \rangle \Downarrow \langle \sigma \rangle$ (see Exercise 42).

Since each rule schema comprises a *recursively enumerable* collection of concrete instance rules, and since our language definition consists of a finite set of rule schemas, by enumerating all the concrete instances of these rules we get a recursively enumerable set of concrete instance rules. Furthermore, since proof trees built with nodes in a recursively enumerable set are themselves recursively enumerable, it follows that the set of proof trees derivable with the proof system in Figure 3.7 is recursively enumerable. In other words, we can find an algorithm that enumerates all the proof trees, in particular one that enumerates all the derivable sequents $C \Downarrow R$. By enumerating all proof trees, given an IMP program p that terminates, one can eventually find the unique state σ such that $\langle p \rangle \Downarrow \langle \sigma \rangle$ is derivable. This simple-minded algorithm may take a very long time and a huge amount of resources, but it is theoretically important to understand that it can be done.

It can be shown that there is no algorithm, based on proof derivation like above or on anything else, which takes as input an IMP program and says whether it terminates or not (see Exercise 43). This follows from the fact that our simple language, due to its while loops and arbitrarily large integers, is Turing-complete. Thus, if one were able to decide termination of programs in our language then one would also be able to decide termination of Turing machines, contradicting one of the basic undecidable problems, the *halting problem* (see Section 4.2 for more on Turing machines).

An interesting observation here is that non-termination of a program corresponds to *lack of proof*, and that the latter is not decidable in many interesting logics. Indeed, in *complete* logics, that is, logics that admit a complete proof system, one can enumerate all the truths. However, in general there is not much one can do about non-truths, because the enumeration algorithm will loop forever when run on a non-truth. In decidable logics one can enumerate both truths and non-truths; clearly, decidable logics are not powerful enough for our task of defining programming languages, precisely because of the halting problem argument above.

3.2.3 Big-Step SOS in Rewriting Logic

Due to its straightforward recursive nature, big-step semantics is typically easy to represent in other formalisms and also easy to translate into interpreters for the defined languages in any programming language. (The difficulty with big-step semantics is to actually give big-step semantics to complex constructs, as discussed in Section 3.10.) It should therefore come at no surprise to the reader that one can associate a conditional rewrite rule to each big-step rule and hereby obtain a rewriting logic theory that faithfully captures the big-step definition.

In this section we first show that any big-step operational semantics BigStep can be mechanically translated into a rewriting logic theory $\mathcal{R}_{\textsc{BigStep}}$ in such a way that the corresponding derivation relations are step-for-step equivalent, that is, $\textsc{BigStep} \vdash C \Downarrow R$ if and only if $\mathcal{R}_{\textsc{BigStep}} \vdash \mathcal{R}_{C \Downarrow R}$, where $\mathcal{R}_{C \Downarrow R}$ is the corresponding syntactic translation of the big-step sequent $C \Downarrow R$ into a (one-step) rewrite rule. Second, we apply our generic translation technique on the big-step operational semantics $\textsc{BigStep}(\text{IMP})$ and obtain a rewriting logic semantics of IMP that is step-for-step equivalent to the original big-step semantics of IMP. Finally, we show how $\mathcal{R}_{\textsc{BigStep}(\text{Imp})}$ can be seamlessly defined in Maude, thus yielding an interpreter for IMP essentially for free.

Faithful Embedding of Big-Step SOS into Rewriting Logic

To define our translation generically, we need to make some assumptions about the existence of an algebraic axiomatization of configurations. More precisely, as also explained in Section 2.5, we assume that for any parametric term t (can be a configuration, a condition, etc.), the term $\bar{t}$ is an equivalent algebraic variant of t of appropriate sort. For example, by "parametric" configuration we mean a configuration that may possibly make use of parameters, such as $a \in AExp$, $\sigma \in State$, etc.; by "equivalent" algebraic variant we mean a term of sort *Configuration* over an appropriate signature of configurations like the one that we defined for IMP in Section 3.2.1 (see Figure 3.6); moreover, each parameter in C gets replaced by a *variable* of corresponding sort in $\overline{C}$. Similarly, a side condition of a rule can be seen as "parametric", in that it constrains some or all of the parameters involved in the rule; its "equivalent" algebraic variant is an appropriate term of sort *Bool*. Consider, for example, the side condition "$\sigma(x)$ defined" of the rules (BigStep-Lookup) and (BigStep-Asgn) in Figure 3.7; its algebraic variant is the term $defined(\sigma, X)$ of *Bool* sort, where σ and X are variables of sorts *State* and *Id*, respectively (see also Section 3.1.2).

Consider now a general-purpose big-step rule of the form

$$\frac{C_1 \Downarrow R_1 \qquad C_2 \Downarrow R_2 \qquad \dots \qquad C_n \Downarrow R_n}{C \Downarrow R} \ [\text{if } condition]$$

where C, C_1, C_2, ..., C_n are configurations holding fragments of program together with all the needed semantic components, R, R_1, R_2, ..., R_n are result configurations, and *condition* is some optional side condition. As one may expect, we translate it into the following rewriting logic rule:

$$\overline{C} \to \overline{R} \ \ \textbf{if} \ \ \overline{C_1} \to \overline{R_1} \ \wedge \ \overline{C_2} \to \overline{R_2} \ \wedge \ \dots \ \wedge \ \overline{C_n} \to \overline{R_n} \ [\wedge \ \overline{condition}].$$

Therefore, the big-step SOS rule premises and side conditions are both turned into conditions of the corresponding rewrite rule. The sequent premises become rewrites in the condition, while the side conditions become simple Boolean checks.

We make the reasonable assumption that configurations in BigStep are not nested.

Theorem 2. ***(Faithful embedding of big-step SOS into rewriting logic)*** *For any big-step operational semantics definition* BigStep*, and any* BigStep *appropriate configuration C and result configuration R, the following equivalence holds*

$$\text{BigStep} \ \vdash C \Downarrow R \iff \mathcal{R}_{\text{BigStep}} \vdash \overline{C} \to^1 \overline{R},$$

where $\mathcal{R}_{\text{BigStep}}$ is the rewriting logic semantic definition obtained from BigStep *by translating each rule in* BigStep *as above. (Recall from Section 2.7 that $\to^1$ is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewriting logic. Also, as C and R are parameter-free—parameters only appear in rules—, $\overline{C}$ and $\overline{R}$ are ground terms.)*

The non-nestedness assumption on configurations in BigStep guarantees that the resulting rewrite rules in $\mathcal{R}_{\text{BigStep}}$ only apply at the top of the term they rewrite. Since one typically perceives parameters as variables anyway, the only apparent difference between BigStep and $\mathcal{R}_{\text{BigStep}}$ is the different notational conventions they use ($\to$ instead of $\Downarrow$ and conditional rewrite rules instead of conditional deduction rules). As Theorem 2 shows, there is a one-to-one correspondence also between their corresponding "computations" (or executions, or derivations). Therefore, $\mathcal{R}_{\text{BigStep}}$ *is* the big-step operational semantics BigStep, and *not* an encoding of it.

At our knowledge, there is no rewrite engine[2] that supports the one-step rewrite relation $\rightarrow^1$ (that appears in Theorem 2). Indeed, rewrite engines aim at high-performance implementations of the general rewrite relation $\rightarrow$, which may even involve parallel rewriting (see Section 2.7 for the precise definitions of $\rightarrow^1$ and $\rightarrow$); $\rightarrow^1$ is meaningful only from a theoretical perspective and there is little to no practical motivation for an efficient implementation of it. Therefore, in order to execute the rewrite theory $\mathcal{R}_{\text{BIGSTEP}}$ resulting from the mechanical translation of big-step semantics BIGSTEP, one needs to take some precautions to ensure that $\rightarrow^1$ is actually identical to $\rightarrow$.

A sufficient condition for $\rightarrow^1$ to be the same as $\rightarrow$ is that the configurations C appearing to the left of $\Downarrow$ are always distinct from those to the right of $\Downarrow$. More generally, if one makes sure that result configurations never appear as left-hand sides of rules in $\mathcal{R}_{\text{BIGSTEP}}$, then one is guaranteed that it is never the case that more than one rewrite step will ever be applied on a given configuration.

Corollary 1. *Under the same hypotheses as in Theorem 2, if result configurations never appear as left-hand sides of rules in* $\mathcal{R}_{\text{BIGSTEP}}$*, then*

$$\text{BIGSTEP} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BIGSTEP}} \vdash \overline{C} \rightarrow^1 \overline{R} \iff \mathcal{R}_{\text{BIGSTEP}} \vdash \overline{C} \rightarrow \overline{R}.$$

Fortunately, in our concrete big-step semantics of IMP, BIGSTEP(IMP), the configurations to the left of $\Downarrow$ and the result configurations to the right of $\Downarrow$ are always distinct. Unfortunately, in general that may not always be the case. For example, when we extend IMP with side effects in Section 3.10, the (possibly affected) state also needs to be part of result configurations, so the semantics of integers is going to be given by an unconditional rule of the form $\langle i, \sigma \rangle \Downarrow \langle i, \sigma \rangle$, which after translation becomes the rewrite rule $\langle i, \sigma \rangle \rightarrow \langle i, \sigma \rangle$. This rule will make the rewrite relation $\rightarrow$ not terminate anymore (although the relation $\rightarrow^1$ terminates). There are at least two simple ways to ensure the hypothesis of Corollary 1:

1. It is highly expected that the only big-step rules in BIGSTEP having a result configuration to the left of $\Downarrow$ are unconditional rules of the form $R \Downarrow R$; such rules typically say that a value is already evaluated. If that is the case, then one can simply drop all the corresponding rules $\overline{R} \rightarrow \overline{R}$ from $\mathcal{R}_{\text{BIGSTEP}}$ and the resulting rewrite theory, say $\mathcal{R}'_{\text{BIGSTEP}}$ still has the property $\text{BIGSTEP} \vdash C \Downarrow R \iff \mathcal{R}'_{\text{BIGSTEP}} \vdash \overline{C} \rightarrow \overline{R}$, which is desirable in order to execute the big-step definition on rewrite engines, although the property $\text{BIGSTEP} \vdash C \Downarrow R \iff \mathcal{R}'_{\text{BIGSTEP}} \vdash \overline{C} \rightarrow^1 \overline{R}$ will not hold anymore, because, e.g., even though $R \Downarrow R$ is a rule in BIGSTEP, it is not the case that $\mathcal{R}'_{\text{BIGSTEP}} \vdash \overline{R} \rightarrow^1 \overline{R}$.

2. If BIGSTEP contains pairs $R' \Downarrow R$ where R' and R are possibly different result configurations, then one can apply the following general procedure. Change or augment the syntax of the configurations to the left or to the right of $\Downarrow$, so that those changed or augmented configurations will always be different from the other ones. This is the technique employed in our representation of small-step operational semantics in rewriting logic in Section 3.3. More precisely, we prepend all the configurations to the left of the rewrite relation in $\mathcal{R}_{\text{BIGSTEP}}$ with a circle $\circ$, e.g., $\circ C \rightarrow R$, with the intuition that the circled configurations are *active*, while the other ones are *inactive*.

[2] Maude's `rewrite[1]` command does not inhibit the transitive closure of the rewrite relation, it only stops the rewrite engine on a given term after *one* application of a rule on that term; however, many (transitive) applications of rules are allowed when solving the condition of that rule.

$$
\begin{array}{rcl}
\langle I, \sigma\rangle \to \langle I\rangle & & \\
\langle X, \sigma\rangle \to \langle \sigma(X)\rangle & \textbf{if} & \sigma(X) \neq \bot \\
\langle A_1 \texttt{ + } A_2, \sigma\rangle \to \langle I_1 \ +_{Int}\ I_2\rangle & \textbf{if} & \langle A_1, \sigma\rangle \to \langle I_1\rangle \ \wedge\ \langle A_2, \sigma\rangle \to \langle I_2\rangle \\
\langle A_1 \texttt{ / } A_2, \sigma\rangle \to \langle I_1 \ /_{Int}\ I_2\rangle & \textbf{if} & \langle A_1, \sigma\rangle \to \langle I_1\rangle \ \wedge\ \langle A_2, \sigma\rangle \to \langle I_2\rangle \ \wedge\ I_2 \neq 0 \\
\langle A_1 \texttt{ <= } A_2, \sigma\rangle \to \langle I_1 \ \leq_{Int}\ I_2\rangle & \textbf{if} & \langle A_1, \sigma\rangle \to \langle I_1\rangle \ \wedge\ \langle A_2, \sigma\rangle \to \langle I_2\rangle \\
\langle T, \sigma\rangle \to \langle T\rangle & & \\
\langle \texttt{not } B, \sigma\rangle \to \langle \texttt{false}\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{true}\rangle \\
\langle \texttt{not } B, \sigma\rangle \to \langle \texttt{true}\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{false}\rangle \\
\langle B_1 \texttt{ and } B_2, \sigma\rangle \to \langle \texttt{false}\rangle & \textbf{if} & \langle B_1, \sigma\rangle \to \langle \texttt{false}\rangle \\
\langle B_1 \texttt{ and } B_2, \sigma\rangle \to \langle T\rangle & \textbf{if} & \langle B_1, \sigma\rangle \to \langle \texttt{true}\rangle \ \wedge\ \langle B_2, \sigma\rangle \to \langle T\rangle \\
\langle \texttt{skip}, \sigma\rangle \to \langle \sigma\rangle & & \\
\langle X \texttt{ := } A, \sigma\rangle \to \langle \sigma[I/X]\rangle & \textbf{if} & \langle A, \sigma\rangle \to \langle I\rangle \ \wedge\ \sigma(X) \neq \bot \\
\langle S_1 \texttt{;} S_2, \sigma\rangle \to \langle \sigma_2\rangle & \textbf{if} & \langle S_1, \sigma\rangle \to \langle \sigma_1\rangle \ \wedge\ \langle S_2, \sigma_1\rangle \to \langle \sigma_2\rangle \\
\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \to \langle \sigma_1\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{true}\rangle \ \wedge\ \langle S_1, \sigma\rangle \to \langle \sigma_1\rangle \\
\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \to \langle \sigma_2\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{false}\rangle \ \wedge\ \langle S_2, \sigma\rangle \to \langle \sigma_2\rangle \\
\langle \texttt{while } B \texttt{ do } S, \sigma\rangle \to \langle \sigma\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{false}\rangle \\
\langle \texttt{while } B \texttt{ do } S, \sigma\rangle \to \langle \sigma'\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{true}\rangle \ \wedge\ \langle S \texttt{;while } B \texttt{ do } S, \sigma\rangle \to \langle \sigma'\rangle \\
\langle \texttt{ var } Xl \texttt{ ; } S\rangle \to \langle \sigma\rangle & \textbf{if} & \langle S, Xl \mapsto 0\rangle \to \langle \sigma\rangle
\end{array}
$$

Figure 3.8: $\mathcal{R}_{\text{BIGSTEP(IMP)}}$: the big-step SOS of IMP in rewriting logic.

Regardless of how the desired property $\text{BIGSTEP} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BIGSTEP}} \vdash \overline{C} \to \overline{R}$ is ensured, note that, unfortunately, $\mathcal{R}_{\text{BIGSTEP}}$ lacks the main strengths of rewriting logic that make it a good formalism for concurrency: in rewriting logic, rewrite rules can apply under any context and in parallel. Indeed, the rules of $\mathcal{R}_{\text{BIGSTEP}}$ can only apply at the top, sequentially.

Big-Step SOS of IMP in Rewriting Logic

Figure 3.8 gives the rewriting logic theory $\mathcal{R}_{\text{BIGSTEP(IMP)}}$ that is obtained by applying the procedure above to the big-step semantics of IMP, BIGSTEP(IMP), in Figure 3.7. We have used the rewriting logic convention that variables start with upper-case letters. For the state variable, we used σ, that is, a larger σ symbol. Note how the three side conditions that appear in the proof system in Figure 3.7 turned into normal conditions of rewrite rules. In particular, the two side conditions saying that $\sigma(x)$ is defined became the algebraic term $\sigma(X) \neq \bot$ of Boolean sort.

The following corollary of Theorem 2 and Corollary 1 establishes the faithfulness of the representation of the big-step operational semantics of IMP in rewriting logic:

Corollary 2. $\text{BIGSTEP(IMP)} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BIGSTEP(IMP)}} \vdash \overline{C} \to \overline{R}$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system BIGSTEP(IMP) and generic rewriting logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{BIGSTEP(IMP)}}$, so the two are faithfully equivalent.

```
mod IMP-CONFIGURATIONS-BIGSTEP is including IMP-SYNTAX + STATE .
  sort Configuration .
  op <_,_> : AExp State -> Configuration .
  op   <_> : Int -> Configuration .
  op <_,_> : BExp State -> Configuration .
  op   <_> : Bool -> Configuration .
  op <_,_> : Stmt State -> Configuration .
  op   <_> : State -> Configuration .
  op   <_> : Pgm -> Configuration .
endm

mod IMP-SEMANTICS-BIGSTEP is including IMP-CONFIGURATIONS-BIGSTEP .
  var X : Id .  var Xl : List{Id} .  var Sigma Sigma' Sigma1 Sigma2 : State .
  var I I1 I2 : Int .  var T : Bool .
  var A A1 A2 : AExp .  var B B1 B2 : BExp .  var S S1 S2 : Stmt .

  rl < I,Sigma > => < I > .
 crl < X,Sigma > => < Sigma(X) >
  if Sigma(X) =/=Bool undefined .
 crl < A1 + A2,Sigma > => < I1 +Int I2 >
  if < A1,Sigma > => < I1 > /\ < A2,Sigma > => < I2 > .
 crl < A1 / A2,Sigma > => < I1 /Int I2 >
  if < A1,Sigma > => < I1 > /\ < A2,Sigma > => < I2 > /\ I2 =/=Bool 0 .

  rl < T,Sigma > => < T > .
 crl < A1 <= A2,Sigma > => < I1 <=Int I2 >
  if < A1,Sigma > => < I1 > /\ < A2,Sigma > => < I2 > .
 crl < not B,Sigma > => < false >
  if < B,Sigma > => <  true > .
 crl < not B,Sigma > => <  true >
  if < B,Sigma > => < false > .
 crl < B1 and B2,Sigma > => < false >
  if < B1,Sigma > => < false > .
 crl < B1 and B2,Sigma > => < T >
  if < B1,Sigma > => < true > /\ < B2,Sigma > => < T > .

  rl < skip,Sigma > => < Sigma > .
 crl < X := A,Sigma > => < Sigma[I / X] >
  if < A,Sigma > => < I > /\ Sigma(X) =/=Bool undefined .
 crl < S1 ; S2,Sigma > => < Sigma2 >
  if < S1,Sigma > => < Sigma1 > /\ < S2,Sigma1 > => < Sigma2 > .
 crl < if B then S1 else S2,Sigma > => < Sigma1 >
  if < B,Sigma > => < true  > /\ < S1,Sigma > => < Sigma1 > .
 crl < if B then S1 else S2,Sigma > => < Sigma2 >
  if < B,Sigma > => < false > /\ < S2,Sigma > => < Sigma2 > .
 crl < while B do S,Sigma > => < Sigma >
  if < B,Sigma > => < false > .
 crl < while B do S,Sigma > => < Sigma' >
  if < B,Sigma > => < true > /\ < S ; while B do S,Sigma > => < Sigma' > .

 crl < var Xl ; S > => < Sigma >
  if < S,(Xl |-> 0) > => < Sigma > .
endm
```

Figure 3.9: The big-step SOS of IMP in Maude, including the definition of configurations.

☆ Maude Definition of IMP Big-Step SOS

Figure 3.9 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{BIGSTEP(IMP)}}$ in Figure 3.8, including a representation of the algebraic signature in Figure 3.6 for configurations as needed for big-step SOS. The Maude module `IMP-SEMANTICS-BIGSTEP` in Figure 3.9 is executable, so Maude, through its rewriting capabilities, yields an interpreter for IMP; for example, the command

```
rewrite < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are irrelevant here, so they were replaced by "..."):

```
rewrites: 5118 in ... cpu (... real) (... rewrites/second)
result Configuration: < n |-> 0 , s |-> 5050 >
```

The obtained IMP interpreter actually has acceptable performance; for example, all the programs in Figure 3.4 together take a fraction of a second to execute on conventional PCs or laptops.

In fact, Maude needs only one rewrite logic step to rewrite any configuration; in particular,

```
rewrite [1] < sumPgm > .
```

will give the same output as above. Recall from Section 2.8 that Maude performs a potentially exhaustive search to satisfy the rewrites in rule conditions. Thus, a large number of rule instances can be attempted in order to apply one conditional rule, so a `rewrite [1]` command can take a long time; it may not even terminate. Nevertheless, thanks to Theorem 2, Maude's implicit search mechanism in conditions effectively achieves a proof searcher for big-step SOS derivations.

Once one has a rewriting logic definition in Maude, one can use any of the general-purpose tools provided by Maude on that definition; the rewrite engine is only one of them. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
search < sumPgm > =>! Cfg:Configuration .
```

Since our IMP language so far is deterministic, the `search` command will not discover any new behaviors. In fact, the search command will only discover two configurations in total, the original configuration `< sumPgm >` and the result configuration `< n |-> 0 & s |-> 5050 >`. However, as shown in Section 3.5 where we extend IMP with various language features, the `search` command can indeed show all the behaviors of a non-deterministic program (restricted only by the limitations of the particular semantic style employed).

3.2.4 Defining a Type System for IMP Using Big-Step SOS

Big-step SOS is routinely used to define type systems for programming languages, even though in most cases this connection is not made explicit. In this section we demonstrate the use of big-step SOS for defining a type system for IMP, following the same steps as above but more succinctly. Type systems is a broad subject, with many variations and important applications to programming languages. Our intention in this section is twofold: on the one hand we show that big-step SOS is not limited to only defining language semantics, and, on the other hand, we introduce the reader to type systems by means of a very simple example.

The idea underlying big-step SOS definitions of type systems is that a given program or fragment of program in a given type environment reduces, in one big step, to its type. Like states, type

environments are also partial mappings, but from variable names into types instead of values. A common notation for a type judgment is $\Gamma \vdash c : \tau$, where Γ is a type environment, c is a program or fragment, and τ is a type. This type judgment reads "in type environment Γ, program or fragment c has type τ". One can find countless variations of the notation for type judgments in the literature, usually adding more items (pieces of information) to the left of $\vdash$, to its right, or as subscripts or superscripts of it. There is, unfortunately, no well-established notation for all type judgments. Nevertheless, type judgments are special big-step sequents relating two special configurations, one including the givens and the other the results. For example, a simple type judgment $\Gamma \vdash c : \tau$ like above can be regarded as a big-step sequent $\langle c, \Gamma \rangle \Downarrow \langle \tau \rangle$. However, this notation is not preferred.

Figure 3.10 depicts our type system for IMP, which is a nothing but a big-step SOS proof system. We, however, follow the more conventional notation for type judgments discussed above, with one slight change: since in IMP variables are intended to hold only integer values, there is no need for type environments; instead, we replace them by lists of variables, each meant to have the type *int*. Therefore, $xl \vdash c : \tau$ with c and τ as above but with xl a list of variables reads as follows: "when the variables in xl are defined, c has the type τ". We drop the list of variables from the typing judgments of programs, because that would be empty anyway. The big-step SOS rules in Figure 3.10 define the typing policy of each language construct of IMP, guaranteeing all together that a program p types, that is, that $\vdash p : pgm$ is derivable if and only if each construct is used according to its intended typing policy and, moreover, that p declares each variable that is uses. For our simple IMP language, a CFG parser using the syntax defined in Figure 3.1 would already guarantee that each construct is used as intended. Note, however, that the second desired property or our type system (each used variable is declared) is context dependent.

Let us next use the type system in Figure 3.10 to type the program `sumPgm` in Figure 3.4. We split the proof tree in proof subtrees. Note first that using the rules (BigStepTypeSystem-Int) (first level), (BigStepTypeSystem-Asgn) (second level) and (BigStepTypeSystem-Seq) (third level), we can derive the following proof tree, say $tree_1$:

$$tree_1 \quad = \quad \left\{ \quad \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{100} : int} \qquad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{0} : int}}{\texttt{n,s} \vdash (\texttt{n := 100}) : stmt} \qquad \dfrac{\cdot}{\texttt{n,s} \vdash (\texttt{s := 0}) : stmt}}{\texttt{n,s} \vdash (\texttt{n := 100; s := 0}) : stmt} \right.$$

Similarly, using rules (BigStepTypeSystem-Lookup) and (BigStepTypeSystem-Int) (first level), (BigStepTypeSystem-Leq) (second level), and (BigStepTypeSystem-Not) (third level), we can derive the following proof tree, say $tree_2$:

$$tree_2 \quad = \quad \left\{ \quad \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : int} \qquad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{0} : int}}{\texttt{n,s} \vdash (\texttt{n <= 0}) : bool}}{\texttt{n,s} \vdash (\texttt{not(n <= 0)}) : bool} \right.$$

$$xl \vdash i : int \qquad \text{(BigStepTypeSystem-Int)}$$

$$(xl, x, xl') \vdash x : int \qquad \text{(BigStepTypeSystem-Lookup)}$$

$$\frac{xl \vdash a_1 : int \qquad xl \vdash a_2 : int}{xl \vdash a_1 \texttt{ + } a_2 : int} \qquad \text{(BigStepTypeSystem-Add)}$$

$$\frac{xl \vdash a_1 : int \qquad xl \vdash a_2 : int}{xl \vdash a_1 \texttt{ / } a_2 : int} \qquad \text{(BigStepTypeSystem-Div)}$$

$$xl \vdash t : bool \qquad \text{(BigStepTypeSystem-Bool)}$$

$$\frac{xl \vdash a_1 : int \qquad xl \vdash a_2 : int}{xl \vdash a_1 \texttt{ <= } a_2 : bool} \qquad \text{(BigStepTypeSystem-Leq)}$$

$$\frac{xl \vdash b : bool}{xl \vdash \texttt{not } b : bool} \qquad \text{(BigStepTypeSystem-Not)}$$

$$\frac{xl \vdash b_1 : bool \qquad xl \vdash b_2 : bool}{xl \vdash b_1 \texttt{ and } b_2 : bool} \qquad \text{(BigStepTypeSystem-And)}$$

$$xl \vdash \texttt{skip} : stmt \qquad \text{(BigStepTypeSystem-Skip)}$$

$$\frac{(xl, x, xl') \vdash a : int}{(xl, x, xl') \vdash (x \texttt{ := } a) : stmt} \qquad \text{(BigStepTypeSystem-Asgn)}$$

$$\frac{xl \vdash s_1 : stmt \qquad xl \vdash s_2 : stmt}{xl \vdash s_1 \texttt{;} s_2 : stmt} \qquad \text{(BigStepTypeSystem-Seq)}$$

$$\frac{xl \vdash b : bool \qquad xl \vdash s_1 : stmt \qquad xl \vdash s_2 : stmt}{xl \vdash \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 : stmt} \qquad \text{(BigStepTypeSystem-If)}$$

$$\frac{xl \vdash b : bool \qquad xl \vdash s : stmt}{xl \vdash \texttt{while } b \texttt{ do } s : stmt} \qquad \text{(BigStepTypeSystem-While)}$$

$$\frac{xl \vdash s : stmt}{\vdash \texttt{ var } xl \texttt{ ; } s : pgm} \qquad \text{(BigStepTypeSystem-Var)}$$

Figure 3.10: BigStepTypeSystem(IMP) — Type system of IMP using big-step SOS ($xl, xl' \in \mathbf{List}\{Id\}$; $i \in Int$; $x \in Id$; $a, a_1, a_2 \in AExp$; $t \in Bool$; $b, b_1, b_2 \in BExp$; $s, s_1, s_2 \in Stmt$).

Similarly, we can derive the following proof tree, say $tree_3$:

$$tree_3 \;=\; \left\{ \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{s} : int} \quad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : int}}{\texttt{n,s} \vdash (\texttt{s + n}) : int} \quad \dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : int} \quad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{-1} : int}}{\texttt{n,s} \vdash (\texttt{n + -1}) : int}}{\dfrac{\texttt{n,s} \vdash (\texttt{s := s + n}) : stmt \qquad \texttt{n,s} \vdash (\texttt{n := n + -1}) : stmt}{\texttt{n,s} \vdash (\texttt{s := s + n; n := n + -1}) : stmt}} \right.$$

Finally, we can now derive the tree that proves that `sumPgm` is well-typed:

$$\dfrac{\dfrac{tree_1 \qquad \dfrac{tree_2 \qquad tree_3}{\texttt{n,s} \vdash (\texttt{while not(n <= 0) do (s := s + n; n := n + -1)}) : stmt}}{\texttt{n,s} \vdash (\texttt{n := 100; s := 0; while not(n <= 0) do (s := s + n; n := n + -1)}) : stmt}}{\vdash (\texttt{var n,s; n := 100; s := 0; while not(n <= 0) do (s := s + n; n := n + -1)}) : pgm}$$

A major role of a type system is to filter out a set of programs which are obviously wrong. Unfortunately, it is impossible to filter out precisely those programs which would execute erroneously. For example, note that a division is considered type safe whenever its two arguments type to integers, but no check is being made on whether the denominator is 0 or not. Indeed, statically checking whether an expression has a certain value at a certain point in a program is an undecidable problem. Also, no check is being made on whether a detected type error is reachable or not (if unreachable, the detected type error will never show up at runtime). Statically checking whether a certain point in a program is reachable is also an undecidable problem. One should therefore be aware of the fact that in general a type system may allow programs which run into errors when executed and, moreover, that it may reject programs which would execute correctly.

Figure 3.11 shows the straightforward translation of the big-step SOS in Figure 3.10 into a rewriting logic theory, following the general technique described in Section 3.2.3. This translation is based on the obvious reinterpretation of type judgments as big-step SOS sequents mentioned above. The following configurations are used in the rewrite theory in Figure 3.11:

- $\langle a, xl \rangle$ grouping arithmetic expressions a and variable lists xl;
- $\langle b, xl \rangle$ grouping Boolean expressions b and variable lists xl;
- $\langle s, xl \rangle$ grouping statements s and variable lists xl;
- $\langle p \rangle$ holding programs p;
- $\langle \tau \rangle$ holding types τ, which can be *int*, *bool*, *stmt*, or *pgm*.

By Corollary 1 we have that a program p is well-typed, that is, $\vdash p : pgm$ is derivable with the proof system in Figure 3.10, if and only if $\mathcal{R}_{\text{BigStepTypeSystem(IMP)}} \vdash \langle p \rangle \to \langle pgm \rangle$.

☆ Maude Definition of a Type System for IMP using Big-Step SOS

Figure 3.12 shows the Maude representation of the rewrite theory $\mathcal{R}_{\text{BigStepTypeSystem(IMP)}}$ in Figure 3.11, including a representation of the algebraic signature for the needed configurations. The Maude module `IMP-TYPE-SYSTEM-BIGSTEP` in Figure 3.12 is executable, so Maude, through its rewriting capabilities, yields a type checker for IMP; for example, the command

$$
\begin{array}{rl}
\langle I, Xl\rangle \to \langle int\rangle & \\
\langle X, (Xl, X, Xl')\rangle \to \langle int\rangle & \\
\langle A_1 \texttt{ + } A_2, Xl\rangle \to \langle int\rangle & \textbf{if } \langle A_1, Xl\rangle \to \langle int\rangle \ \wedge\ \langle A_2, Xl\rangle \to \langle int\rangle \\
\langle A_1 \texttt{ / } A_2, Xl\rangle \to \langle int\rangle & \textbf{if } \langle A_1, Xl\rangle \to \langle int\rangle \ \wedge\ \langle A_2, Xl\rangle \to \langle int\rangle \\
\langle A_1 \texttt{ <= } A_2, Xl\rangle \to \langle bool\rangle & \textbf{if } \langle A_1, Xl\rangle \to \langle int\rangle \ \wedge\ \langle A_2, Xl\rangle \to \langle int\rangle \\
\langle T, Xl\rangle \to \langle bool\rangle & \\
\langle \texttt{not } B, Xl\rangle \to \langle bool\rangle & \textbf{if } \langle B, Xl\rangle \to \langle bool\rangle \\
\langle B_1 \texttt{ and } B_2, Xl\rangle \to \langle bool\rangle & \textbf{if } \langle B_1, Xl\rangle \to \langle bool\rangle \ \wedge\ \langle B_2, Xl\rangle \to \langle bool\rangle \\
\langle \texttt{skip}, Xl\rangle \to \langle stmt\rangle & \\
\langle X \texttt{ := } A, (Xl, X, Xl')\rangle \to \langle stmt\rangle & \textbf{if } \langle A, (Xl, X, Xl')\rangle \to \langle int\rangle \\
\langle S_1 \texttt{ ; } S_2, Xl\rangle \to \langle stmt\rangle & \textbf{if } \langle S_1, Xl\rangle \to \langle stmt\rangle \ \wedge\ \langle S_2, Xl\rangle \to \langle stmt\rangle \\
\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, Xl\rangle \to \langle stmt\rangle & \textbf{if } \langle B, Xl\rangle \to \langle bool\rangle \ \wedge\ \langle S_1, Xl\rangle \to \langle stmt\rangle \ \wedge\ \langle S_2, Xl\rangle \to \langle stmt\rangle \\
\langle \texttt{while } B \texttt{ do } S, Xl\rangle \to \langle stmt\rangle & \textbf{if } \langle B, Xl\rangle \to \langle bool\rangle \ \wedge\ \langle S, Xl\rangle \to \langle stmt\rangle \\
\langle \texttt{ var } Xl \texttt{ ; } S\rangle \to \langle pgm\rangle & \textbf{if } \langle S, Xl\rangle \to \langle stmt\rangle
\end{array}
$$

Figure 3.11: $\mathcal{R}_{\text{BigStepTypeSystem(IMP)}}$: the type system of IMP using big-step SOS in rewriting logic.

```
rewrite < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are irrelevant here, so they were replaced by "..."):

```
rewrites: 19 in ... cpu (... real) (... rewrites/second)
result Configuration: < pgm >
```

A type system is generally expected to be deterministic. Nevertheless, implementations of it (particularly rewrite-based ones) may mistakenly be non-deterministic (non-confluent; see Section 2.6). To gain confidence in the determinism of the Maude definition in Figure 3.12, one may exhaustively search for all possible behaviors yielded by the type checker:

```
search < sumPgm > =>! Cfg:Configuration .
```

As expected, this finds only one solution. This Maude definition of IMP's type checker is very simple and one can easily see that it is confluent (it is orthogonal—see Section 2.6), so the search is redundant. However, the search command may be useful for testing more complex type systems.

3.2.5 Notes

Big-step structural operational semantics (big-step SOS) was introduced under the name *natural semantics* by Kahn [38] in 1987. Even though he introduced it in the limited context of defining Mini-ML, a simple pure (no side effects) version of the ML language, Kahn's aim was to propose natural semantics as a "unified manner to present different aspects of the semantics of programming languages, such as dynamic semantics, static semantics and translation" (Section 1.1 in [38]). Kahn's original notation for big-step sequents was $\sigma \vdash a \Rightarrow i$, with the meaning that a evaluates to i in state (or environment) σ. Kahn, like many others after him (including ourselves; e.g., Section 3.2.4), took the freedom to using a different notation for type judgments, namely $\Gamma \vdash c : \tau$, where Γ is a

```
mod IMP-TYPES is
  sort Type .
  ops int bool stmt pgm : -> Type .
endm

mod IMP-TYPE-SYSTEM-CONFIGURATIONS-BIGSTEP is including IMP-SYNTAX + IMP-TYPES .
  sort Configuration .
  op <_,_> : AExp List{Id} -> Configuration .
  op <_,_> : BExp List{Id} -> Configuration .
  op <_,_> : Stmt List{Id} -> Configuration .
  op   <_> : Pgm -> Configuration .
  op   <_> : Type -> Configuration .
endm

mod IMP-TYPE-SYSTEM-BIGSTEP is including IMP-TYPE-SYSTEM-CONFIGURATIONS-BIGSTEP .
  var X : Id .  var Xl Xl' : List{Id} .  var I : Int .  var T : Bool .
  var A A1 A2 : AExp .  var B B1 B2 : BExp .  var S S1 S2 : Stmt .

  rl < I,Xl > => < int > .
  rl < X,(Xl,X,Xl') > => < int > .
 crl < A1 + A2,Xl > => < int >
  if < A1,Xl > => < int > /\ < A2,Xl > => < int > .
 crl < A1 / A2,Xl > => < int >
  if < A1,Xl > => < int > /\ < A2,Xl > => < int > .

  rl < T,Xl > => < bool > .
 crl < A1 <= A2,Xl > => < bool >
  if < A1,Xl > => < int > /\ < A2,Xl > => < int > .
 crl < not B,Xl > => < bool >
  if < B,Xl > => < bool > .
 crl < B1 and B2,Xl > => < bool >
  if < B1,Xl > => < bool > /\ < B2,Xl > => < bool > .

  rl < skip,Xl > => < stmt > .
 crl < X := A,(Xl,X,Xl') > => < stmt >
  if < A,(Xl,X,Xl') > => < int > .
 crl < S1 ; S2,Xl > => < stmt >
  if < S1,Xl > => < stmt > /\ < S2,Xl > => < stmt > .
 crl < if B then S1 else S2,Xl > => < stmt >
  if < B,Xl > => < bool > /\ < S1,Xl > => < stmt > /\ < S2,Xl > => < stmt > .
 crl < while B do S,Xl > => < stmt >
  if < B,Xl > => < bool > /\ < S,Xl > => < stmt > .

 crl < var Xl ; S > => < pgm >
  if < S,Xl > => < stmt > .
endm
```

Figure 3.12: The type-system of IMP using big-step SOS in Maude, including the definition of types and configurations.

type environment, c is a program or fragment of program, and τ is a type. This colon notation for type judgments was already established in 1987; however, Kahn noticed that the way type systems were defined was a special instance of a more general schema, which he called natural semantics (and which is called big-step SOS here and in many other places). Big-step SOS is very natural when defining pure, sequential and structured languages, so it quickly became very popular. However, Kahn's terminology for "natural" semantics was inspired from its reminiscence to "natural deduction" in mathematical logic, not necessarily from the fact that it is natural to use.

As Kahn himself acknowledged, the idea of using proof systems to capture the operational semantics of programming languages goes back to Plotkin [70, 71] in 1981. Plotkin was the first to coin the terminology *structural operational semantics* (S0S), but what he meant by that was mostly what we call today *small-step* structural operational semantics (small-step SOS). Note, however, that Plotkin in fact used a combination of small-step and big-step SOS, without calling them as such, using the $\rightarrow$ arrow for small-steps and its transitive closure $\rightarrow^*$ for big-steps. We will discuss small-step SOS in depth in Section 3.3. Kahn and others found big-step SOS more natural and convenient than Plotkin's SOS, essentially because it is more abstract and denotational in nature, and one needs fewer rules to define a language semantics.

One of the most notable uses of natural semantics is the formal semantics of Standard ML by Milner *et al.* [58]. Several types of big-step sequents were used in [58], such as $\rho \vdash p \Rightarrow v/f$ for "in environment ρ, sentence p either evaluates to value v or otherwise an error or failure f takes place", and $\sigma, \rho \vdash p \Rightarrow v, \sigma'$ for "in state σ and environment ρ, sentence p evaluates to v and the resulting state is σ'", and $\rho, v \vdash m \Rightarrow v'/f$ for "in environment ρ, a match m either evaluates to v' or otherwise failure f", among many others. After more than twenty years of natural semantics, it is now common wisdom that big-step semantics is inappropriate as a rigorous formalism for defining languages with complex features such as exceptions or concurrency. To give a reasonably compact and readable definition of Standard ML in [58], Milner *et al.* had to make several informal notational conventions, such as a "state convention" to avoid having to mention the state in every rule, and an "exception convention" to avoid having to more than double the number of rules for the sole purpose of supporting exceptions. As rightfully noticed by Mosses [62], such conventions are not only adhoc and language specific, but may also lead to erroneous definitions. Section 3.5 illustrates in detail the limitations of big-step operational semantics, both with respect to its incapacity of defining certain rather simple language features and with respect to inconvenience in using it (for example, due to its lack of modularity). One of the common uses of natural semantics these days is to define static semantics of programming languages and calculi, such as type systems (see Section 3.2.4).

Hennessy [36] (1990) and Winskel [98] (1993) are perhaps the first textbooks proposing big-step SOS in teaching programming language semantics. They define big-step SOS for several simple languages, including ones similar to the IMP language presented in this chapter. Hennessy [36] defines languages incrementally, starting with a small core and then adding new features one by one, highlighting a major problem with big-step SOS: its lack of modularity. Indeed, the big-step SOS of a language is entirely redefined several times in [36] as new features are added to the language, because adding new features requires changes in the structure of judgments. For example, some big-step SOS judgments in [36] evolve from $a \Rightarrow i$, to $\sigma \vdash a \Rightarrow i$, to $D, \sigma \vdash i$ during the language design experiment, where a is an expression, i an integer, σ a state (or environment), and D a set of function definitions.

While the notations of Hennessy [36] and of Milner *et al.* [58] are somehow reminiscent of original Kahn's notation, Winskel [98] uses a completely different notation. More precisely, he prefers to use

big-step sequents of the form $\langle a, \sigma \rangle \to i$ instead of $\sigma \vdash a \Rightarrow i$. There seems to be, unfortunately, no uniform and/or broadly accepted notation for SOS sequents in the literature, be they big-step or small-step. As already explained earlier in this section, for the sake of uniformity at least throughout this book, we will make and effort to consider sequents of the form $C \Downarrow R$ in our big-step SOS definitions, where C and R are configurations. Similarly, we will make an effort to use the notation $C \to C'$ for small-step sequents (see Section 3.3). We will make it explicit when we deviate from our uniform notation, explaining how the temporary notation relates to the uniform one, as we did in Section 3.2.4.

Big-step SOS is the semantic approach which is probably the easiest to implement in any language or to represent in any computational logic. There are countless approaches to implementing or encoding big-step SOS in various languages or logics, which we cannot enumerate here. We only mention rewriting-based ones which are directly related to the approach followed in this book. Vardejo and Martí-Oliet [95] proposed big-step SOS implementations in Maude for several languages, including Hennessy's languages [36] and Kahn's Mini-ML [38]. Vardejo and Martí-Oliet were mainly interested in demonstrating the strengths of Maude 2 to give executable semantics to concrete languages, rather than in proposing general representations of big-step SOS into rewriting logic that work for any language. In particular, their big-step sequents mimicked those in the original big-step SOS, e.g., they used sequents having the syntax $\sigma \vdash a \Rightarrow i$ in their Maude implementation of Kahn's Mini-ML. Besides Vardejo and Martí-Oliet, several other authors used rewriting logic and Maude to define and implement language semantics for languages or calculi following a small-step approach. These are discussed in Section 3.3.4; we here only want to emphasize that most of those can likely be adapted into big-step SOS definitional or implementation styles, because big-step SOS can be regarded as a special case of small-step SOS, one in which the small step is "big".

3.2.6 Exercises

Prove the following exercises, all referring to the IMP big-step SOS in Figure 3.7.

Exercise 39. *Change the rule* BigStep-Div *so that division short-circuits when a_1 evaluates to 0. (Hint: may need to replace it with two rules, like for the semantics of conjunction).*

Exercise 40. *Change the big-step semantics of the* IMP *conjunction so that it is not short-circuited.*

Exercise 41. *Add an* error *state and modify the big-step semantics in Figure 3.7 to allow derivations of sequents of the form $\langle s, \sigma \rangle \Downarrow \langle \mathsf{error} \rangle$ or $\langle p \rangle \Downarrow \langle \mathsf{error} \rangle$ when s evaluated in state σ or when p evaluated in the initial state performs a division by zero.*

Exercise 42. *Prove that the transition relation defined by the* BigStep(IMP) *proof system in Figure 3.7 is* ***deterministic****, that is:*

- *If* $\text{BigStep(IMP)} \vdash \langle a, \sigma \rangle \Downarrow \langle i \rangle$ *and* $\text{BigStep(IMP)} \vdash \langle a, \sigma \rangle \Downarrow \langle i' \rangle$ *then* $i = i'$*;*
- *If* $\text{BigStep(IMP)} \vdash \langle b, \sigma \rangle \Downarrow \langle t \rangle$ *and* $\text{BigStep(IMP)} \vdash \langle b, \sigma \rangle \Downarrow \langle t' \rangle$ *then* $t = t'$*;*
- *If s terminates in σ then there is a unique σ' such that* $\text{BigStep(IMP)} \vdash \langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle$*;*
- *If p terminates then there is a unique σ such that* $\text{BigStep(IMP)} \vdash \langle p \rangle \Downarrow \langle \sigma \rangle$*.*

Prove the same results above for the proof system detecting division-by-zero as in Exercise 41.

Exercise 43. *Show that there is no algorithm, based on the big-step proof system in Figure 3.7 or on anything else, which takes as input an* IMP *program and says whether it terminates* ***or not****.*