Programming Language Semantics

A Rewriting Approach

Grigore Roșu

University of Illinois at Urbana-Champaign

Chapter 2

Background and Preliminaries

2.6 Complete Partial Orders and the Fixed-Point Theorem

This section introduces a fixed-point theorem for complete partial orders. Our main use of this theorem is to give denotational semantics to iterative (in Section 3.4) and to recursive (in Section 4.8) language constructs. Complete partial orders with bottom elements are at the core of both domain theory and denotational semantics, often identified with the notion of "domain" itself.

2.6.1 Posets, Upper Bounds and Least Upper Bounds

Here we recall basic notions of partial and total orders, such as upper bounds and least upper bounds, and discuss several examples and counter-examples.

Definition 14. *A **partial order**, or a **poset** (from partial order set) $(D, \sqsubseteq)$ consists of a set D and a binary relation $\sqsubseteq$ on D, written as an infix operation, which is*

- ***reflexive**, that is, $x \sqsubseteq x$ for any $x \in D$;*
- ***transitive**, that is, for any $x, y, z \in D$, $x \sqsubseteq y$ and $y \sqsubseteq z$ imply $x \sqsubseteq z$; and*
- ***anti-symmetric**, that is, if $x \sqsubseteq y$ and $y \sqsubseteq x$ for some $x, y \in D$ then $x = y$.*

*A partial order $(D, \sqsubseteq)$ is called a **total order** when $x \sqsubseteq y$ or $y \sqsubseteq x$ for any $x, y \in D$.*

Here are some examples of partial or total orders:

- $(\mathcal{P}(S), \subseteq)$ is a partial order, where $\mathcal{P}(S)$ is the powerset of a set S, that is, the set of subsets of S.
- $(Nat, \leq)$, the set of natural numbers ordered by "less than or equal to", is a total order.
- $(Nat, \geq)$, the set of natural numbers ordered by "larger than or equal to", is a total order.
- $(Nat \cup \{\infty\}, \leq)$, the naturals plus infinity, where infinity is larger than any number, is a total order.
- $(Int, \leq)$, the set of integer numbers, is a total order.
- $(Real, \leq)$, the set of real numbers, is a total order.
- $(S, =)$, a flat set S where the only partial ordering is the identity, is a partial order.
- $(S \cup \{\bot\}, \leq^S_\bot)$, a set S extended with a bottom element $\bot$ with the property that for any $a, b \in S \cup \{\bot\}$, $a \leq^S_\bot b$ if and only if $a = b$ or $a = \bot$, is a partial order. Such extensions of sets with bottom elements are common in denotational semantics (Section 3.4), where the intuition for the $\bot$ is *undefined*. They are commonly written more simply as $S_\bot$ and are often called *primitive* or *flat* domains.
- A partial order which is particularly important for denotational semantics (Section 3.4) is $(A \rightharpoonup B, \preceq)$, the set of partial functions from A to B partially ordered by the *informativeness relation* $\preceq$: given partial functions $f, g : A \rightharpoonup B$, f is "less informative than or as informative as" g, written $f \preceq g$, iff for any $a \in A$, it is either the case that $f(a)$ is not defined or both $f(a)$ and $g(a)$ are defined and $f(a) = g(a)$.

- If $(S_1, \leq_1)$ and $(S_2, \leq_2)$ are partial or total orders and $S_1 \cap S_2 = \emptyset$ then $(S_1 \cup S_2, \leq_1 \cup \leq_2)$ is a partial order, where $a(\leq_1 \cup \leq_2)b$ if and only if there is some $i \in \{1, 2\}$ such that $a, b \in S_i$ and $a \leq_i b$. This union partial order is typically not a total order; it is a total order only when $(S_1, \leq_1)$ is total and $S_2 = \emptyset$, or the other way around. More generally, if $\{(S_i, \leq_i)\}_{i \in I}$ is a family of partial orders such that $S_i \cap S_j = \emptyset$ for any $i \neq j \in I$, then $\bigcup_{i \in I}(S_i, \leq_i) \stackrel{\text{def}}{=} (\bigcup_{i \in I} S_i, \bigcup_{i \in I} \leq)$ is a partial order, where $a \ (\bigcup_{i \in I} \leq_i) \ b$ if and only if there is some $i \in I$ such that $a, b \in S_i$ and $a \leq_i b$.

- If $\{(S_i, \leq_i)\}_{i \in I}$ is a family of partial orders then $\prod_{i \in I}(S_i, \leq_i) \stackrel{\text{def}}{=} (\prod_{i \in I} S_i, \prod_{i \in I} \leq_i)$ is also a partial order, where $\{a_i\}_{i \in I} \ (\prod_{i \in I} \leq_i) \ \{b_i\}_{i \in I}$ iff $a_i \leq_i b_i$ for all $i \in I$ (i.e., the product partial order is defined component-wise).

Definition 15. *Given partial order $(D, \sqsubseteq)$ and a set of elements $X \subseteq D$, an element $p \in D$ is called an **upper bound** of X iff $x \sqsubseteq p$ for any $x \in X$. Furthermore, $p \in D$ is called the **least upper bound (LUB)** of X, written $\sqcup X$, iff p is an upper bound and for any other upper bound q of X it is the case that $p \sqsubseteq q$.*

Upper bounds and least upper bounds may not always exist. For example, if $D = X = \{x, y\}$ and $\sqsubseteq$ is the identity relation, then X has no upper bounds. Least upper bounds may not exist even though upper bounds exist. For example, if $D = \{a, b, c, d, e\}$ and $\sqsubseteq$ is defined by $a \sqsubseteq c$, $a \sqsubseteq d$, $b \sqsubseteq c$, $b \sqsubseteq d$, $c \sqsubseteq e$, $d \sqsubseteq e$, then any subset of D admits upper bounds, but the set $\{a, b\}$ does not have a LUB. Due to the anti-symmetry property, least upper bounds are unique when they exist, which justifies the notation $\sqcup X$ in Definition 15.

As an analogy with mathematical analysis, we can think of the LUB of a set of elements as their *limit*. For example, in $(Real, \leq)$ the LUB of the set $\{n/(n+1) \mid n \in Nat\}$ coincides with its limit in the mathematical analysis sense, namely 1. Note that $([0, 1), \leq)$ does not admit LUBs for all its subsets, e.g., $\{n/(n+1) \mid n \in Nat\}$ does not have a LUB. One might think that this happens because $([0, 1), \leq)$ does not even admit upper bounds for all its subsets. If we add the interval $(4, 5]$ to the set above, for example, then the resulting (total) order admits upper bounds for any of its subsets (e.g., 5 is such an upper bound) but the set $\{n/(n+1) \mid n \in Nat\}$ still does not have a LUB. However, if we add the interval $[4, 5]$ to the original interval $[0, 1)$ then the resulting (total) order admits LUBs for all its subsets; for example, the LUB of $\{n/(n+1) \mid n \in Nat\}$ is 4.

2.6.2 Complete Partial Orders

In the decimal representation of real numbers, completeness means that any infinite string of decimal digits is actually the decimal representation for some real number. In mathematical analysis in general, a set of real numbers is complete iff the limit of any converging sequence of real numbers is also in the set. For example, the interval $[0, 1)$ open in 1 is not complete, because the sequence $1 - 1/n$ for $n > 0$ converges to 1 but 1 is not in the set. On the other hand, the interval $[0, 1]$ is complete. Many particular variants of completeness that appear in the literature are in fact instances a general notion or completeness for partial orders.

Definition 16. *Given a poset $(D, \sqsubseteq)$, a **chain** in D is an infinite sequence $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots$ of elements in D, also written using the set notation as $\{d_n \mid n \in Nat\}$. Such a chain is called **stationary** when there is some $n \in Nat$ such that $d_m = d_{m+1}$ for all $m \geq n$.*

If D is finite then any chain is stationary. More generally, if for a given $x \in D$ there are only finitely many elements $y \in D$ such that $x \sqsubseteq y$, then any chain containing x is stationary.

Definition 17. *A poset $(D, \sqsubseteq)$ is called a **complete partial order (CPO)** iff any of its chains has a LUB. $(D, \sqsubseteq)$ is said to **have bottom** iff it has a minimal element. Such an element is typically written $\bot$, and the poset*

with bottom $\bot$ is written $(D, \sqsubseteq, \bot)$. CPOs with bottom are also called ***bottomed complete partial orders****. If $\{d_n \mid n \in Nat\}$ is a chain in $(D, \sqsubseteq)$, then we also let $\bigsqcup_{n \in Nat} d_n$, or simply $\sqcup d_n$, denote its LUB, $\sqcup\{d_n \mid n \in Nat\}$.*

Let us follow up on the examples underneath Definition 14 and discuss which of them are CPOs with botom and which are not:

- $(\mathcal{P}(S), \subseteq, \emptyset)$ is a bottomed CPO.
- $(Nat, \leq)$ has bottom 0 but is not complete: the chain $0 \leq 1 \leq 2 \leq \cdots \leq n \leq \cdots$ has no upper bound.
- $(Nat, \geq)$ is a CPO but has no bottom.
- $(Nat \cup \{\infty\}, \leq, 0)$ is a CPO with bottom 0: it is a CPO because any chain is either stationary, in which case its LUB is obvious, or is unbounded by any natural number, in which case ∞ is its LUB.
- $(Int, \leq)$ is not a CPO and has no bottom.
- $(S, =)$ is always a CPO, and it has bottom if and only if S has only one element.
- $(S \cup \{\bot\}, \leq_\bot^S, \bot)$ is a bottomed CPO. It is common to use the same notation $S_\bot$, used for the partial order $(S \cup \{\bot\}, \leq_\bot^S)$, to also denote its corresponding bottomed CPO $(S \cup \{\bot\}, \leq_\bot^S, \bot)$. It is interesting and particularly important in denotational semantics to note that the primitive domain $S_\bot$ is equivalent to the set of partial functions $\{*\} \rightharpoonup S$, where $\{*\}$ is the singleton set.
- $(A \rightharpoonup B, \preceq, \bot)$, the set of partial functions $A \rightharpoonup B$ ordered by the informativeness relation $\preceq$, is a CPO with bottom $\bot : A \rightharpoonup B$, the partial function which is undefined in each element of A.
- If $\{(S_i, \leq_i, \bot)\}_{i \in I}$ is a family of bottomed CPOs with same bottom $\bot$ such that $S_i \cap S_j = \{\bot\}$ for any $i \neq j \in I$, then $\bigcup_{i \in I}(S_i, \leq_i, \bot)$ is also a bottomed CPO.
- If $\{(S_i, \leq_i)\}_{i \in I}$ is a family of bottomed CPOs then $\prod_{i \in I}(S_i, \leq_i)$ is also a bottomed CPO.

All the CPOs of interest discussed in this book have a bottom element and, with a few minor exceptions in the next section, all the results that we prove for CPOs require bottom elements. Consequently, unless otherwise explicitly stated, we assume that all CPOs are bottomed CPOs.

2.6.3 Monotone and Continuous Functions

The same way various notions of completeness encountered in the literature are instances of the general notion of complete partial orders in Section 2.6.2, various particular notions of monotone and of continuous functions are instances of homonymous notions defined generally for partial orders and CPOs.

Definition 18. *If $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ are two posets and $\mathcal{F} : D \to D'$ is a function, then $\mathcal{F}$ is called* ***monotone*** *if and only if $\mathcal{F}(x) \sqsubseteq' \mathcal{F}(y)$ for any $x, y \in D$ with $x \sqsubseteq y$. If $\mathcal{F}$ is monotone, then we simply write $\mathcal{F} : (D, \sqsubseteq) \to (D', \sqsubseteq')$. Let $Mon((D, \sqsubseteq), (D', \sqsubseteq'))$ denote the set of monotone functions from $(D, \sqsubseteq)$ to $(D', \sqsubseteq')$.*

Monotone functions *preserve chains*, that is, $\{\mathcal{F}(d_n) \mid n \in Nat\}$ is a chain in $(D', \sqsubseteq')$ whenever $\{d_n \mid n \in Nat\}$ is a chain in $(D, \sqsubseteq)$. Moreover, if $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ are CPOs then for any chain $\{d_n \mid n \in Nat\}$ in $(D, \sqsubseteq)$, we have $\sqcup \mathcal{F}(d_n) \sqsubseteq' \mathcal{F}(\sqcup d_n)$. Indeed, since $\mathcal{F}$ is monotone and since $d_n \sqsubseteq \sqcup d_n$ for each $n \in Nat$, it follows

that $\mathcal{F}(d_n) \sqsubseteq' \mathcal{F}(\sqcup d_n)$ for each $n \in Nat$. Therefore, $\mathcal{F}(\sqcup d_n)$ is an upper bound for the chain $\{\mathcal{F}(d_n) \mid n \in Nat\}$. The rest follows because $\sqcup\mathcal{F}(d_n)$ is the LUB of $\{\mathcal{F}(d_n) \mid n \in Nat\}$.

Note that $\mathcal{F}(\sqcup d_n) \sqsubseteq' \sqcup\mathcal{F}(d_n)$ does not hold in general. Let, for example, $(D, \sqsubseteq)$ be $(Nat \cup \{\infty\}, \leq)$, $(D', \sqsubseteq')$ be $(\{0, \infty\}, 0 \leq \infty)$, and $\mathcal{F}$ be the monotone function taking any natural number to 0 and ∞ to ∞. For the chain $\{n \mid n \in Nat\}$, note that $\sqcup n = \infty$, so $\mathcal{F}(\sqcup n) = \infty$. On the other hand, the chain $\{\mathcal{F}(n) \mid n \in Nat\}$ is stationary in 0, so $\sqcup\mathcal{F}(n) = 0$. Therefore, $\mathcal{F}(\sqcup n) = \infty \not\leq 0 = \sqcup\mathcal{F}(n)$.

Recall that one can think of the LUB of a chain as the limit of that chain. The following definition is inspired from the analogous notion of continuous function in mathematical analysis, which is characterized by the property of preserving limits:

Definition 19. *Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be CPOs. A monotone function $\mathcal{F} : (D, \sqsubseteq) \to (D', \sqsubseteq')$ is **continuous** iff $\mathcal{F}(\sqcup d_n) \sqsubseteq' \sqcup\mathcal{F}(d_n)$, which is equivalent to $\sqcup\mathcal{F}(d_n) = \mathcal{F}(\sqcup d_n)$, for any chain $\{d_n \mid n \in Nat\}$ in $(D, \sqsubseteq)$. Let $Cont((D, \sqsubseteq), (D', \sqsubseteq'))$ denote the set of continuous functions from $(D, \sqsubseteq)$ to $(D', \sqsubseteq')$. This notation extends seamlessly to bottomed CPO: $Cont((D, \sqsubseteq, \bot), (D', \sqsubseteq', \bot'))$.*

If we let $S_\bot$ denote the bottomed CPO extension of the set S as discussed in Section 2.6.2, then it can be shown that there is a bijective correspondence between the set of partial functions in $A \rightharpoonup B$ and the set of continuous (total) functions in $Cont(A_\bot, B_\bot)$ (see Exercises 37 and 38).

Note that continuous functions in $Cont((D, \sqsubseteq, \bot), (D', \sqsubseteq', \bot'))$ need not take $\bot$ to $\bot'$ in general. In fact, as seen shortly in Section 2.6.4, the interesting continuous functions do not have that property.

Proposition 12. *$Cont((D, \sqsubseteq, \bot), (D', \sqsubseteq', \bot'))$ can be endowed with a bottomed CPO structure, where*

- *Its partial order is defined as $f \preceq g$ iff $f(d) \leq g(d)$ for all $d \in D$;*
- *Its bottom element is the continuous function taking each element $d \in D$ to $\bot'$.*

Proof. ... □

Recall from Section 2.6.2 that for any sets A and B, the set of partial functions $A \rightharpoonup B$ can be organized as a bottomed CPO, $(A \rightharpoonup B, \preceq, \bot)$. Moreover, it can be shown (see Exercise 39) that $(A \rightharpoonup B, \preceq, \bot)$ and $(Cont(A_\bot, B_\bot), \preceq, \bot)$ are in fact isomorphic bottomed CPOs. This way, Proposition 12 allows us to associate a bottomed CPO to any higher-order type. For example, the type `(int -> int) -> (int -> bool)` of functions taking functions from *Int* to *Int* into functions from *Int* to *Bool* can be associated the bottomed CPO $Cont(Cont(Int_\bot, Int_\bot), Cont(Int_\bot, Bool_\bot))$. This uniform view of types as CPOs will allows us in Section 4.8 to use the fixed-point theorem discussed next to elegantly give denotational semantics to higher-order languages with recursive functions of arbitrary types.

Moreover, we can organize all the CPOs above together as one big CPO. Indeed, given any family of bottomed CPOs $\{B_i\}_{i \in I}$ (thought of as CPOs corresponding to some arbitrary set of basic types), let $\mathcal{HO}$ (from higher-order model) be the smallest set closed under the following:

- $B_i \in \mathcal{HO}$ for any $i \in I$;
- If $X, Y \in \mathcal{HO}$ then $Cont(X, Y) \in \mathcal{HO}$.

In Section 2.7 we will see that $\mathcal{HO}$ can be regarded as a cartesian-closed-category. Now let us additionally assume that all CPOs B_i have the same bottom element $\bot$ and that $B_i \cap B_j = \{\bot\}$ for any $i, j \in I$. Also, assume that we use the same $\bot$ element as bottom in all the CPOs $Cont(X, Y) \in \mathcal{HO}$. Since the union of CPOs whose pairwise intersection is their common bottom element $\{\bot\}$ is also a CPO with bottom $\{\bot\}$ (Section 2.6.2), we conclude that the union of all the CPOs in $\mathcal{HO}$ is also a CPO with bottom $\{\bot\}$.

2.6.4 The Fixed-Point Theorem

Any monotone function $\mathcal{F} : (D, \sqsubseteq, \bot) \to (D, \sqsubseteq, \bot)$ defined on a bottomed CPO to itself admits an implicit and important chain, namely $\bot \sqsubseteq \mathcal{F}(\bot) \sqsubseteq \mathcal{F}^2(\bot) \sqsubseteq \cdots \sqsubseteq \mathcal{F}^n(\bot) \sqsubseteq \cdots$, where $\mathcal{F}^n$ denotes n compositions of $\mathcal{F}$ with itself. The next theorem is a key result, which has major implications in many areas of mathematics and computer science, in particular in denotational semantics (Section 3.4):

Theorem 12. ***(The fixed-point theorem)*** *Let $(D, \sqsubseteq, \bot)$ be a bottomed CPO, let $\mathcal{F} : (D, \sqsubseteq, \bot) \to (D, \sqsubseteq, \bot)$ be a continuous function, and let $fix(\mathcal{F})$ be the LUB of the chain $\{\mathcal{F}^n(\bot) \mid n \in Nat\}$. Then $fix(\mathcal{F})$ is the least fix-point of $\mathcal{F}$.*

Proof. We first show that $fix(\mathcal{F})$ is a fix-point of $\mathcal{F}$:

$$\begin{array}{rcl} \mathcal{F}(fix(\mathcal{F})) & = & \mathcal{F}(\bigsqcup_{n \in Nat} \mathcal{F}^n(\bot)) \\ & = & \bigsqcup_{n \in Nat} \mathcal{F}^{n+1}(\bot) \\ & = & \bigsqcup_{n \in Nat} \mathcal{F}^n(\bot) \\ & = & fix(\mathcal{F}). \end{array}$$

Next we show that $fix(\mathcal{F})$ is the least fix-point of $\mathcal{F}$. Let d be another fix-point of $\mathcal{F}$, that is, $\mathcal{F}(d) = d$. We can show by induction that $\mathcal{F}^n(\bot) \sqsubseteq d$ for any $n \in Nat$: first note that $\mathcal{F}^0(\bot) = \bot \sqsubseteq d$; assume $\mathcal{F}^n(\bot) \sqsubseteq d$ for some $n \in Nat$; since $\mathcal{F}$ is monotone, it follows that $\mathcal{F}(\mathcal{F}^n(\bot)) \sqsubseteq \mathcal{F}(d) = d$, that is, $\mathcal{F}^{n+1}(\bot) \sqsubseteq d$. Thus d is an upper bound of the chain $\{\mathcal{F}^n(\bot) \mid n \in Nat\}$, so $fix(\mathcal{F}) \sqsubseteq d$. □

Many recursive definitions in mathematics and computer science are given informally, but they are more subtle than they appear to be. The fixed-point theorem can be used to formally argue that such definitions are indeed correct. Consider, for example, the following common definition of the factorial:

$$f(n) = \begin{cases} 1 & \text{if } \ n = 0 \\ n * f(n-1) & \text{if } \ n > 0 \end{cases}$$

How can we know whether such a mathematical object, i.e., a function f satisfying the above property, actually exists and is unique, as tacitly assumed? To see that such a concern is justified, replace "$n * f(n-1)$ if $n > 0$" by "$n * f(n-1)$ if $n > 1$" in the above; now there are infinitely many functions satisfying the above property. Or replace $n * f(n-1)$ by $n * f(n+1)$; now there is no function with the property above. According to the fixed-point theorem, since the function $\mathcal{F}$ defined on the set of partial functions $Nat \rightharpoonup Nat$ to itself as

$$\mathcal{F}(g)(n) = \begin{cases} 1 & \text{if } \ n = 0 \\ n * g(n-1) & \text{if } \ n > 0 \text{ and } g(n-1) \text{ defined} \\ \text{undefined} & \text{if } \ n > 0 \text{ and } g(n-1) \text{ undefined} \end{cases}$$

is continuous, it has a least fixed point. We thus can take $f = fix(\mathcal{F})$, and get

$$f(n) = \mathcal{F}(f)(n) = \begin{cases} 1 & \text{if } \ n = 0 \\ n * f(n-1) & \text{if } \ n > 0 \text{ and } f(n-1) \text{ defined} \\ \text{undefined} & \text{if } \ n > 0 \text{ and } f(n-1) \text{ undefined} \end{cases}$$

One can easily show (by induction) that f is defined everywhere, so the above is equivalent to the original definition of f. Since f is total, it is the *unique* fixed point of $\mathcal{F}$: any fixed-point f' of $\mathcal{F}$ obeys $f \preceq f'$, so f' is also defined everywhere and equal to f. See also Exercise 40.

Another application of the fixed-point theorem is to show why and how recursively defined languages admit unique solution. Any context-free language over a possibly infinite alphabet, or set of terminals, is the least fixed point of some continuous operator on the power set of the set of words over the given alphabet. Let for instance the alphabet be $A = \mathit{Var} \cup \mathit{Int} \cup \{+, -, *\}$, where *Int* is the set of integers and *Var* is a set of variables, and consider the following context-free grammar giving the syntax for arithmetic expressions:

$$\mathit{Exp} \ ::= \ \mathit{Var} \ \mid \mathit{Int} \ \mid \mathit{Exp} + \mathit{Exp} \ \mid -\mathit{Exp} \ \mid \mathit{Exp} * \mathit{Exp}$$

There are many ways to describe or construct the language of arithmetic expressions corresponding to the grammar above, which we do not discuss here. What we want to highlight is that it can also be described as the least fixed point of the following continuous function, where A^* is the set of finite words with letters in A:

$\mathcal{F} : (\mathcal{P}(A^*), \subseteq, \emptyset) \to (\mathcal{P}(A^*), \subseteq, \emptyset)$, where

$$\mathcal{F}(L) \ = \ \mathit{Var} \ \cup \ \mathit{Int} \ \cup \ L\{+\}L \ \cup \ \{-\}L \ \cup \ L\{*\}L.$$

We used the notation $L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. Notice that the iterations $\mathcal{F}(\emptyset)$, $\mathcal{F}^2(\emptyset)$, ... correspond to the one-step, two-steps, ... derivations applying the grammar's productions. See Exercise 41.

We shall later see that the fixed-point theorem above can also be used to give denotational semantics both to iterative (in Section 3.4) and to recursive (in Section 4.8.3) constructs. Fixed-points in general and the theorem above in particular have many applications in both computer science and mathematics.

2.6.5 Equational Framework for Fixed-Points

As seen above, fixed-points can be complex mathematical objects, such as functions or infinite sets of words, which may require complex means to define, understand or visualize. In many cases the only option is to *experiment* with the fixed-point, such as to evaluate it on a given input when it is a function, or to check whether it contains a certain element when it is a set. If one wants to formally reason about or to compute fixed points, then one needs to formally support, explicitly or implicitly, the body of mathematics involved in the definition of the function for which the fixed point is desired. For example, the factorial above requires support for the domain of integers, at least with multiplications and subtraction, as well as support for the domain $\mathit{Int} \rightharpoonup \mathit{Int}$ of partial functions from integers to integers (in order to define $\mathcal{F}$). Similarly, the CFG language above requires support for sets with union and possibly other operations on them, and for the domain of finite words over a given alphabet, with at least a concatenation operation on them, as well as a combination of the two, where concatenation is extended to sets of words. Even though in theory we assume the entire arsenal of mathematics, in practice we only have a limited fragment of mathematics available for formal definitions of fixed points. In this section we define such a limited framework, but one which is quite common and particularly useful for the application of fixed-points in denotational semantics (Section 3.4).

Figure 2.12 shows a simple equational theory of a minimal framework for formally defining fixed-points. We call it minimal because it only includes support for defining and applying functions, together with a simple extension for defining cases. One will need to add further extensions to this framework in order to support definitions of more complex fixed-points. For defining cases, we assumed that Booleans are already defined. For convenience, from here on we also assume that integers are already defined, together with all the desired operations on both Booleans and integers. To make it clear from which mathematical domain each operation or relational symbol comes, we add the main domain sort as a subscript to the operator or relational symbol. For example, $+_{\mathit{Int}}$ is the addition on integers, $\leq_{\mathit{Int}}$ is their comparison which evaluates to a Boolean, and $=_{\mathit{Int}}$ is their equality test. The functions, their application and their fixed-points have been defined like in untyped call-by-value λ-calculus (see Section 4.5) extended with the recursion μ construct (see Section 4.8),

sorts:

Var_{CPO}, CPO

subsorts:

Var_{CPO}, $Bool$ $<$ CPO

operations:

$\texttt{fun}_{CPO}\ \_\ \texttt{->}\ \_ : Var_{CPO} \times CPO \to CPO$

$\texttt{app}_{CPO}(\_, \_) : CPO \times CPO \to CPO$

$\texttt{fix}_{CPO}\ \_ : CPO \to CPO$

$\texttt{if}_{CPO}(\_, \_, \_) : Bool \times CPO \times CPO \to CPO$

equations:

$\texttt{app}_{CPO}(\texttt{fun}_{CPO}\ V\ \texttt{->}\ C, C') = C[C'/V]$

$\texttt{fix}_{CPO}\ \texttt{fun}_{CPO}\ V\ \texttt{->}\ C = C[(\texttt{fix}_{CPO}\ \texttt{fun}_{CPO}\ V\ \texttt{->}\ C)/V]$

$\texttt{if}_{CPO}(\texttt{true}, C, C') = C$

$\texttt{if}_{CPO}(\texttt{false}, C, C') = C'$

Figure 2.12: Equational framework for CPOs and fixed-points (assuming Booleans and substitution).

using an appropriate notion of substitution (see Section 4.5.3). How these are defined is not important here, so we refer the interested reader to the above-mentioned sections for details. What is important here is that the equational theory in Figure 2.12 gives us a starting point for formally defining fixed-points.

For simplicity, we considered only one major syntactic category in the equational theory in Figure 2.12, namely *CPO*, for complete partial orders. Basic CPOs other than that of Booleans $Bool_\perp$, e.g., the CPO of integers $Int_\perp$, need to be subsorted to *CPO* if one wants to include them. We concentrate on functional domains here, that is, on CPOs whose elements are partial functions, with $\perp$ the partial function which is undefined everywhere. We followed our naming convention and thus we tagged all the newly introduced operations with *CPO*, to distinguish them from possibly homonymous operations from included domains. We implicitly assume that the various domains subsorted to *CPO* are responsible for generating undefinedness (e.g., when a division by zero in the domain of integers is attempted) and appropriately propagating it. The four equations in Figure 2.12 are straightforward, capturing the meaning of the four language constructs in terms of substitution; here we assume substitution given, as explained above.

We can now use the constructs in Figure 2.12 to define fixed-points. For example, the factorial function discussed above can be defined as the term below of sort *CPO*, say `factorial`:

$$\texttt{fix}_{CPO}\ \texttt{fun}_{CPO}\ g\ \texttt{->}\ \texttt{fun}_{CPO}\ n\ \texttt{->}\ \texttt{if}_{CPO}(n =_{Int} 0, 1, \texttt{if}_{CPO}(n >_{Int} 0, n *_{Int} \texttt{app}_{CPO}(g, n -_{Int} 1), \perp))$$

Using the provided equations, one can prove, for example, that `factorial`(3) = 6. The only difference between our definition of factorial above and the fixed-point definition right after Theorem 12 is that we assumed the common mathematical convention that undefinedness is propagated implicitly through the various domain operations, so we did not propagate it explicitly. See Exercise 42 for another example.

The equational theory in Figure 2.12 can be easily defined in rewrite logic systems like Maude, provided that a generic substitution is available. For example, Appendix A.1 shows one way to do it, together with several extensions not discussed here. Once such a Maude definition of the mathematical domain is available, we can use it to calculate and experiment with fixed-points, in particular to execute recursive functions like the factorial above and, more importantly, later (Section 3.4) to execute denotational semantics of programming languages, thus obtaining interpreters for the defined languages directly from their formal semantics.

2.6.6 Exercises

Exercise 37. *Recall from Section 2.6.2 that given a set S, we let $S_\perp$ denote the poset $(S \cup \{\perp\}, \leq^S_\perp)$ with $a \leq^S_\perp b$ if and only if $a = b$ or $a = \perp$, for any $a, b \in S \cup \{\perp\}$. Let A and B be two arbitrary sets.*

1. *Characterize the monotone functions $\mathcal{F} \in Mon(A_\perp, B_\perp)$ with the property that $\mathcal{F}(\perp) \neq \perp$;*
2. *Are there any monotone functions $\mathcal{F} \in Mon(A_\perp, B_\perp)$ such that $\mathcal{F}(a) = \perp$ for some $\perp \neq a \in A$?*
3. *For this item only, suppose that A and B are finite:*
   (a) *How many partial functions are there in $A \rightharpoonup B$?*
   (b) *How many (not necessarily monotone) total functions are there in $A_\perp \to B_\perp$?*
   (c) *How many monotone functions are there in $Mon(A_\perp, B_\perp)$?*
4. *Show that there is a bijective correspondence between partial functions in $A \rightharpoonup B$ and monotone functions in $Mon(A_\perp, B_\perp)$. Give an example of a total function in $A_\perp \to B_\perp$ which does not correspond to a partial function in $A \rightharpoonup B$.*

Exercise 38. *(See also Exercise 37). For any sets A and B, $Mon(A_\perp, B_\perp) = Cont(A_\perp, B_\perp)$ and, moreover, this set of total functions bijectively corresponds to the set of partial functions $A \rightharpoonup B$.*

Exercise 39. *For any sets A and B, $(Cont(A_\perp, B_\perp), \leq, \perp)$ and $(A \rightharpoonup B, \leq, \perp)$ are isomorphic bottomed CPOs.*

Exercise 40. *Prove that the function $\mathcal{F}$ associated to the factorial function discussed right after Theorem 12 satisfies the hypothesis of Theorem 12, in particular that it is continuous. What if we replace "$n * g(n-1)$ if $n > 0$" by "$n * g(n-1)$ if $n > 1$", or $n * g(n-1)$ by $n * g(n+1)$?*

Exercise 41. *Prove that the function $\mathcal{F}$ associated to the context-free grammar discussed after Theorem 12 satisfies the hypothesis of Theorem 12, in particular that it is continuous. Describe a general procedure to associate such a function to any context-free grammar, so that its least fixed point is precisely the language of the grammar.*

Exercise 42. *Define the famous Ackermann function*

$$a(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ a(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ a(m-1, a(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

as a fixed-point and then represent it in the formal domain language in Figure 2.12 in two different ways: first as a (curried) function taking one argument and producing a function taking another argument and then producing an integer results, and second as a function taking a pair argument. For the latter, extend the equational theory in Figure 2.12 with a pair construct and corresponding projection operations.

2.6.7 Notes

Complete partial orders play a central role in theoretical computer science and especially in denotational semantics (Section 3.4) and in domain theory (started with the seminal paper by Dana Scott [71]). Some of the original work in these fields was done using *complete latices* (i.e., partial orders in which all subsets have both a supremum and an infimum), which also admit fixed-point theorems. Nevertheless, CPOs are generally

considered to have better properties than complete latices, particularly in the context of hight-order languages, so these days both denotational semantics and domain theory build upon CPOs instead of complete latices.

Complete partial orders are sometimes called *directed-complete partial orders*, or ω-complete partial orders, and abbreviated DCPOs or ω-CPOs. Also, the notion of continuity as we used in this section is sometimes called *Scott continuity*, named after Dana Scott. There are many fixed-point theorems in mathematics and in computer science; the variant we used in this section is typically attributed to Bronisław Knaster and Alfred Tarski, and is called the Knaster-Tarski fixed-point theorem. At the time this book was being written, there was a conference totally dedicated to fixed points, called the *International Conference on Fixed Point Theory and Its Applications*.

Chapter 3

Conventional Executable Semantics

Int	::=	the domain of (unbounded) integer numbers, with usual operations on them
Bool	::=	the domain of Booleans
Id	::=	standard identifiers
AExp	::=	*Int*
	|	*Id*
	|	*AExp* + *AExp*
	|	*AExp* / *AExp*
BExp	::=	*Bool*
	|	*AExp* <= *AExp*
	|	! *BExp*
	|	*BExp* && *BExp*
Block	::=	{}
	|	{ *Stmt* }
Stmt	::=	*Block*
	|	*Id* = *AExp* ;
	|	*Stmt Stmt*
	|	`if (` *BExp* `)` *Block* `else` *Block*
	|	`while (` *BExp* `)` *Block*
Pgm	::=	`int` **List{***Id***};** *Stmt*

Figure 3.1: Syntax of IMP, a small imperative language, using algebraic BNF.

3.1 IMP: A Simple Imperative Language

To illustrate the various semantic styles discussed in this chapter, we have chosen a small imperative language, called IMP, whose syntax is inspired from C and Java. In fact, if we wrap an IMP program in a `main(){...}` function the we get a valid C program. IMP has arithmetic expressions which include the domain of arbitrarily large integer numbers, Boolean expressions, assignment statements, conditional statements, while loop statements, and sequential composition of statements. Statements can be grouped in blocks surrounded with curly brackets, and the branches of the conditional and the loop body are required to be blocks. All variables used in an IMP program need to be declared at the beginning of the program, can only hold integer values (for simplicity, IMP has no Boolean variables), and are initialized with default value 0.

3.1.1 IMP Syntax

We here define the syntax of IMP, first using the Backus-Naur form (BNF) notation for context-free grammars and then using the alternative and completely equivalent mixfix algebraic notation (see Section 2.1.3). The latter is in general more appropriate for semantic developments of a language.

IMP Syntax as a Context-Free Grammar

Figure 3.1 shows the syntax of IMP using the algebraic BNF notation. In this book we implicitly assume parentheses as part of any syntax, without defining them explicitly. Parentheses can be freely used for grouping, to increase clarity and/or to avoid ambiguity in parsing. For example, with the syntax in Figure 3.1, `(x + 3) / y` is a well-formed IMP arithmetic expression.

The only algebraic feature in the IMP syntax in Figure 3.1 is the use of **List{***Id***}** for variable declarations (last production), which in this case is clear: one can declare a comma-separated list of variables. To stay more conventional in notation, we refrained from replacing the productions *Stmt* ::= {} | *Stmt Stmt* with the algebraic production *Stmt* ::= $\mathbf{List}^{\{\}}_{;}${*Stmt*} which captures the idea of statement sequentialization more naturally. Moreover, our syntax for statement sequential composition allows ambiguous parsing. Indeed, if $s_1, s_2, s_3 \in$ *Stmt* then $s_1\ s_2\ s_3$ can be parsed either as $(s_1\ s_2)\ s_3$ or as $s_1\ (s_2\ s_3)$. However, the semantics of statement sequential composition will be such that the parsing ambiguity is irrelevant (but that may not always be the case). It may be worthwhile pointing out that one should not get tricked by thinking that different parsings mean different evaluation orders. In our case here, both $(s_1\ s_2)\ s_3$ and $s_1\ (s_2\ s_3)$ will proceed by evaluating the three statements in order. The difference between the two is that the former will first evaluate $s_1\ s_2$ and then s_3, while the latter will first evaluate s_1 and then $s_2\ s_3$; in either case, s_1, s_2 and s_3 will end up being evaluated in the same order: first s_1, then s_2, and then s_3.

The IMP language constructs have their usual imperative meaning. For diversity and demonstration purposes, when giving the various semantics of IMP we will assume that + is *non-deterministic* (it evaluates the two subexpressions in any order, possibly interleaving their corresponding evaluation steps), / is non-deterministic and *partial* (it will stuck the program when a division by zero takes place), <= is *left-right sequential* (it first evaluates the left subexpression and then the right subexpression), and that && is left-right sequential and *short-circuited* (it first evaluates the left subexpression and then it conditionally evaluates the right only if the left evaluated to true).

One of the main reasons for which arithmetic language constructs like + above are allowed to be non-deterministic in language semantic definitions is because one wants to allow flexibility in how the language is implemented, and not because these operations are indeed intended to have fully non-deterministic, or random, behaviors in all implementations. In other words, their non-determinism is to a large extent an artifact of their intended *underspecification*. Some language manuals actually state explicitly that one should not rely on the order in which the arguments of language constructs are evaluated. In practice, it is considered to be programmers' responsibility to write their programs in such a way that one does not get different behaviors when the arguments are evaluated in different orders.

To better understand the existing semantic approaches and to expose some of their limitations, Section 3.5 discusses extensions of IMP with expression side effects (a variable increment operation), with abrupt termination (a halt statement), with dynamic threads and join synchronization, with local variable declarations, as well as with all of these together; the resulting language is called IMP++. The extension with side effects, in particular, makes the evaluation strategies of +, <= and && semantically relevant.

Each semantical approach relies on some basic mathematical infrastructure, such as integers, Booleans, etc., because each semantic definition reduces the semantics of the language constructs to those domains. We will assume available any needed mathematical domains, as well as basic operations on them which are clearly tagged (e.g., $+_{Int}$ for the addition of integer numbers, etc.) to distinguish them from homonymous operations which are language constructs. Unless otherwise stated, we assume no implementation-specific restrictions in our mathematical domains; for example, we assume integer numbers to be arbitrarily large rather than representable on 32 bits, etc. We can think of the underlying domains used in language semantics as parameters of the semantics; indeed, changing the meaning of these domains changes the meaning of all language semantics using them. We also assume that each mathematical domain is endowed with a special element, written $\perp$ for all domains to avoid notational clutter, corresponding to *undefined* values of that domain. Some of these mathematical domains are defined in Chapter 2; appropriate references will be given when such domains are used.

sorts:

Int, *Bool*, *Id*, *AExp*, *BExp*, *Block*, *Stmt*, *Pgm*

subsorts:

$$\begin{array}{rcl} \mathit{Int}, \mathit{Id} & < & \mathit{AExp} \\ \mathit{Bool} & < & \mathit{BExp} \\ \mathit{Block} & < & \mathit{Stmt} \end{array}$$

operations:

$$\begin{array}{rcl} \_ \texttt{+} \_ & : & \mathit{AExp} \times \mathit{AExp} \to \mathit{AExp} \\ \_ \texttt{/} \_ & : & \mathit{AExp} \times \mathit{AExp} \to \mathit{AExp} \\ \_ \texttt{<=} \_ & : & \mathit{AExp} \times \mathit{AExp} \to \mathit{BExp} \\ \texttt{!} \_ & : & \mathit{BExp} \to \mathit{BExp} \\ \_ \texttt{\&\&} \_ & : & \mathit{BExp} \times \mathit{BExp} \to \mathit{BExp} \\ \{\} & : & \to \mathit{Block} \\ \{ \_ \} & : & \mathit{Stmt} \to \mathit{Block} \\ \_ \texttt{=} \_ \texttt{;} & : & \mathit{Id} \times \mathit{AExp} \to \mathit{Stmt} \\ \_ \_ & : & \mathit{Stmt} \times \mathit{Stmt} \to \mathit{Stmt} \\ \texttt{if(\_)} \_ \texttt{else} \_ & : & \mathit{BExp} \times \mathit{Block} \times \mathit{Block} \to \mathit{Stmt} \\ \texttt{while(\_)} \_ & : & \mathit{BExp} \times \mathit{Block} \to \mathit{Stmt} \\ \texttt{int} \_ \texttt{;} \_ & : & \mathbf{List\{}\mathit{Id}\mathbf{\}} \times \mathit{Stmt} \to \mathit{Pgm} \end{array}$$

Figure 3.2: Syntax of IMP as an algebraic signature.

We take the freedom to tacitly use the following naming conventions for meta or mathematical variables[1] ranging over IMP-specific terms throughout the remainder of this chapter: $x, X \in \mathit{Id}$; $a, A \in \mathit{AExp}$; $b, B \in \mathit{BExp}$; $s, S \in \mathit{Stmt}$; $i, I \in \mathit{Int}$; $t, T \in \mathit{Bool}$; $p, P \in \mathit{Pgm}$. Any of these can be primed or indexed.

IMP Syntax as an Algebraic Signature

Following the relationship between the CFG and the mixfix algebraic notations explained in Section 2.1.3, the BNF syntax in Figure 3.1 can be associated the entirely equivalent algebraic signature in Figure 3.2 with one (mixfix) operation per production: the terminals mixed with underscores form the name of the operation and the non-terminals give its arity. This signature is easy to define in any rewrite engine or theorem prover; moreover, it can also be defined as a data-type or corresponding structure in any programming language. We next show how it can be defined in Maude.

★ Definition of IMP Syntax in Maude

Using the Maude notation for algebraic signatures, the algebraic signature in Figure 3.2 can yield the Maude syntax module in Figure 3.3. We have additionally picked some appropriate precedences and formatting attributes for the various language syntactic constructs (see Section 2.5.6 for more details on Maude and the meaning of these attributes).

[1] Recall that we use an *italic* font for such variables, in contrast to the `typewriter` font that we use for code (including program variable identifiers, integers, operation symbols, etc.). For example, if we write $x, \mathtt{x} \in \mathit{Id}$ then we mean an arbitrary identifier that x refers to, and *the concrete* identifier `x`. The latter can appear in programs, while the former cannot. The former is mainly used to define semantics or state properties of the language.

```
mod IMP-SYNTAX is including PL-INT + PL-BOOL + PL-ID .
--- AExp
  sort AExp .  subsorts Int Id < AExp .
  op _+_ : AExp AExp -> AExp [prec 33 gather (E e) format (d b o d)] .
  op _/_ : AExp AExp -> AExp [prec 31 gather (E e) format (d b o d)] .
--- BExp
  sort BExp .  subsort Bool < BExp .
  op _<=_ : AExp AExp -> BExp  [prec 37 format (d b o d)] .
  op !_ : BExp -> BExp [prec 53 format (b o d)] .
  op _&&_ : BExp BExp -> BExp [prec 55 gather (E e) format (d b o d)] .
--- Block and Stmt
  sorts Block Stmt .  subsort Block < Stmt .
  op {} : -> Block [format (b b o)] .
  op {_} : Stmt -> Block [format (d n++i n--i d)] .
  op _=_; : Id AExp -> Stmt [prec 40 format (d b o b o)] .
  op __ : Stmt Stmt -> Stmt [prec 60 gather (e E) format (d ni d)] .
  op if(_)_else_ : BExp Block Block -> Stmt [prec 59 format (b so d d s nib o d)] .
  op while(_)_ : BExp Block -> Stmt [prec 59 format (b so d d s d)] .
--- Pgm
  sort Pgm .
  op int_;_ : List{Id} Stmt -> Pgm [prec 70 format (nb o d ni d)] .
endm
```

Figure 3.3: IMP syntax as an algebraic signature in Maude. This definition assumes appropriate modules PL-INT, PL-BOOL and PL-ID defining corresponding sorts Int, Bool, and Id, respectively.

The module IMP-SYNTAX in Figure 3.3 imports three builtin modules, namely: PL-INT, which we assume it provides a sort Int; PL-BOOL, which we assume provides a sort Bool; and PL-ID which we assume provides a sort Id. We do not give the precise definitions of these modules here, particularly because one may have many different ways to define them. In our examples from here on in the rest of this chapter we assume that PL-INT contains all the integer numbers as constants of sort Int, that PL-BOOL contains the constants true and false of sort Bool, and that PL-ID contains all the letters in the alphabet as constants of sort Id. Also, we assume that the module PL-INT comes equipped with as many builtin operations on integers as needed. To avoid operator name conflicts caused by Maude's operator overloading capabilities, we urge the reader to *not* use the Maude builtin INT and BOOL modules, but instead to overwrite them. Appendix A.1 shows one possible way to do this: we define new modules PL-INT and PL-BOOL "hooked" to the builtin integer and Boolean values but defining only a subset of operations on them and with clearly tagged names to avoid name overloading, e.g., _+Int_, _/Int_, etc.

Recall from Sections 2.4.6 and 2.5.6 that lists, sets, bags, and maps are trivial algebraic structures which can be easily defined in Maude; consequently, we take the freedom to use them without definition whenever needed, as we did with using the sort List{Id} in Figure 3.3.

To test the syntax, one can now parse various IMP programs, such as:

```
Maude> parse
   int n, s ;
   n = 100 ;
   while (!(n <= 0)) {
     s = s + n ;
     n = n + -1 ;
   }
.
```

```
mod IMP-PROGRAMS is including IMP-SYNTAX .
  ops sumPgm collatzPgm countPrimesPgm : -> Pgm .
  ops collatzStmt multiplicationStmt primalityStmt : -> Stmt .
  eq sumPgm = (
     int n, s ;
     n = 100 ;
     while (!(n <= 0)) {
       s = s + n ;
       n = n + -1 ;
     }    ) .

  eq collatzStmt = (
     while (!(n <= 1)) {
       s = s + 1 ;  q = n / 2 ;  r = q + q + 1 ;
       if (r <= n) { n = n + n + n + 1 ; } else { n = q ; }
     }     ) .

  eq collatzPgm = (
     int m, n, q, r, s ;
     m = 10 ;
     while (!(m <= 2)) {
       n = m ;
       m = m + -1 ;
       collatzStmt
     }     ) .

  eq multiplicationStmt = (        --- fast multiplication (base 2) algorithm
     z = 0 ;
     while (!(x <= 0)) {
       q = x / 2 ;
       r = q + q + 1 ;
       if (r <= x) { z = z + y ; } else {}
       x = q ;
       y = y + y ;
     }     ) .

  eq primalityStmt = (
     i = 2 ;  q = n / i ;  t = 1 ;
     while (i <= q && 1 <= t) {
       x = i ;
       y = q ;
       multiplicationStmt
       if (n <= z) { t = 0 ; } else { i = i + 1 ;  q = n / i ; }
     }     ) .

  eq countPrimesPgm = (
     int i, m, n, q, r, s, t, x, y, z ;
     m = 10 ;  n = 2 ;
     while (n <= m) {
       primalityStmt
       if (1 <= t) { s = s + 1 ; } else {}
       n = n + 1 ;
     }     ) .
endm
```

Figure 3.4: IMP programs defined in a Maude module IMP-PROGRAMS.

Now it is a good time to define a module, say `IMP-PROGRAMS`, containing as many IMP programs as one bears to write. Figure 3.4 shows such a module containing several IMP programs. Note that we took advantage of Maude's rewriting capabilities to save space and reuse some of the defined fragments of programs as "macros". The program `sumPgm` calculates the sum of numbers from 1 to 100; since we do not have subtraction in IMP, we decremented the value of `n` by adding -1.

The program `collatzPgm` in Figure 3.4 tests Collatz' conjecture for all numbers from 1 to 10, counting the total number of steps in `s`. The Collatz conjecture, still unsolved, is named after Lothar Collatz (but also known as the $3n+1$ conjecture), who first proposed it in 1937. Take any natural number n. If n is even, divide it by 2 to get $n/2$, if n is odd multiply it by 3 and add 1 to obtain $3n+1$. Repeat the process indefinitely. The conjecture claims that no matter what number you start with, you will always eventually reach 1. Paul Erdös said about the Collatz conjecture: "Mathematics is not yet ready for such problems." While we do not attempt to solve it, we can test it even in a simple language like IMP. It is a good example program to test IMP semantics because it makes use of almost all IMP's language constructs and also has nested loops. The macro `collatzStmt` detaches the check of a single `n` from the top-level loop iterating `n` through all $2 < \texttt{n} \leq \texttt{m}$. Note that, since we do not have multiplication and test for even numbers in IMP, we mimic them using the existing IMP constructs.

Finally, the program `countPrimesPgm` counts all the prime numbers up to `m`. It uses `primalityStmt`, which checks whether `n` is prime or not (writing `t` to 1 or to 0, respectively), and `primalityStmt` makes use of `multiplicationStmt`, which implements a fast base 2 multiplication algorithm. Defining such a module with programs helps us to test the desired language syntax (Maude will report errors if the programs that appear in the right-hand sides of the equations are not parsable), and will also help us later on to test the various semantics that we will define.

3.1.2 IMP State

Any operational semantics of IMP needs some appropriate notion of *state*, which is expected to map program variables to integer values. Moreover, since IMP disallows uses of undeclared variables, it suffices for the state of a given program to only map the declared variables to integer values and stay undefined in the variables which were not declared.

Fortunately, all these desired IMP state operations correspond to conventional mathematical operations on *partial finite-domain functions* from variables to integers in $[Id \rightharpoonup Int]^{finite}$ (see Section 2.1.2) or, equivalently, to structures of sort $\mathbf{Map}\{Id \mapsto Int\}$ defined using equations (see Section 2.4.6 for details on the notation and the equivalence); we let *State* be an alias for the map sort above. From a semantic point of view, the equations defining such map structures are computationally invisible: semantic transitions that are part of various IMP semantics will be performed *modulo* these equations. In other words, state lookup and update operations will not count as computational steps, so they will not interfere with or undesirably modify the intended computational granularity of the defined language.

We let σ, σ', σ_1, etc., range over states. By defining IMP states as partial finite-domain functions $\sigma : Id \rightharpoonup Int$, we have a very natural notion of undefinedness for a variable that has not been declared and thus has not been initialized in a state: variable x is considered *undefined* in a state σ if and only if $x \notin Dom(\sigma)$. We may use the terminology *state lookup* for the operation $\_(\_) : State \times Id \rightharpoonup Int$, the terminology *state update* for the operation $\_[\_/\_] : State \times Int \times Id \to State$, and the terminology *state initialization* for the operation $\_ \mapsto \_ : \mathbf{List}\{Id\} \times Int \to State$.

Recall from Section 2.1.2 that the lookup operation is itself a partial function, because the state to lookup may be undefined in the variable of interest; as usual, we let $\bot$ denote the undefined state and we write as expected $\sigma(x) = \bot$ and $\sigma(x) \neq \bot$ when the state σ is undefined and, respectively, defined in variable x. Recall

```
mod STATE is including PL-INT + PL-ID .
  sort State .

  op _|->_ : List{Id} Int -> State [prec 0] .
  op .State : -> State .
  op _&_ : State State -> State [assoc comm id: .State format(d s s d)] .

  op _(_) : State Id -> Int [prec 0] .          --- lookup
  op _[_/_] : State Int Id -> State [prec 0] .  --- update

  var Sigma : State .  var I I' : Int .  var X X' : Id .  var Xl : List{Id} .

  eq X |-> undefined = .State .                 --- "undefine" a variable in a state

  eq (Sigma & X |-> I)(X) = I .
  eq Sigma(X) = undefined [owise] .

  eq (Sigma &  X |-> I)[I' / X ] = (Sigma & X |-> I') .
  eq Sigma[I / X] = (Sigma & X |-> I) [owise] .

  eq (X,X',Xl) |-> I = X |-> I & X' |-> I & Xl |-> I .
  eq .List{Id} |-> I = .State .
endm
```

Figure 3.5: The IMP state defined in Maude.

also from Section 2.1.2 that the update operation can be used not only to update maps but also to "undefine" particular elements in their domain: $\sigma[\bot/x]$ is the same as σ in all elements different from x and is undefined in x. Finally, recall also from Section 2.1.2 that the initialization operation yields a partial function mapping each element in the first list argument to the element given as second argument. These can be easily defined equationally, following the equational approach to partial finite-domain functions in Section 2.4.6.

★ Definition of IMP State in Maude

Figure 3.5 adapts the generic Maude definition of partial finite-domain functions in Section 2.5.6 for our purpose here: the generic sorts `Source` and `Target` are replaced by `Id` and `Int`, respectively. Recall from Section 2.5.6 that the constant `undefined` has sort `Undefined`, which is a subsort of all sorts corresponding to mathematical domains (e.g., `Int`, `Bool`, etc.). This way, identifiers can be made "undefined" in a state by simply updating them with `undefined` (see the equation dissolving undefined bindings in Figure 3.5).

To avoid overloading the comma "`,`" construct for too many purposes (which particularly may confuse Maude's parser), we took the freedom to rename the associative and commutative construct for states to `&`. The only reason for which we bother to give this obvious module here is because we want the various subsequent semantics of the IMP language, all of them including the module `STATE` in Figure 3.5, to be self-contained and executable in Maude by simply executing all the Maude code in the figures in this chapter.

3.1.3 Notes

The style that we follow in this chapter, namely to pick a simple language and then demonstrate the various language definitional approaches by means of that simple language, is quite common. In fact, we named our language IMP after a similar language introduced by Winskel in his book [87], also called IMP, which

is essentially identical to ours except that it uses a slightly different syntax and does not have variable declarations. For example, Winskel's IMP uses "`:=`" for assignment and "`;`" as statement separator instead of statement terminator, while our IMP's syntax resembles that of common languages like C and Java. Also, since most imperative languages do have variable declarations, we feel it is instructive to include them in our simple language. Winskell gives his IMP a big-step SOS, a small-step SOS, a denotational semantics, and an axiomatic semantics. Later, Nipkow [55] formalized all these semantics of IMP in the Isabelle/HOL proof assistant [56], and used it to formally relate the various semantics, effectively mechanizing most of Winskel's paper proofs; in doing so, Nipkow [55] found several minor errors in Winskel's proofs, thus showing the benefits of mechanization.

Vardejo and Martì-Oliet [83, 84] show how to use Maude to implement executable semantics for several languages following both big-step and small-step SOS approaches. Like us, they also demonstrate how to define different semantics for the same simple language using different styles; they do so both for an imperative language (very similar to our IMP) and for a functional language. Șerbănuță *et al.* [74] use a similar simple imperative language to also demonstrate how to use rewrite logic to define executable semantics. In fact, this chapter is an extension of [74], both in breadth and in depth. For example, we state and prove general faithful rewrite logic representation results for each of the semantic approaches, while [74] did the same only for the particular simple imperative language considered there. Also, we cover new approaches here, such as denotational semantics, which were not covered in [83, 84, 74].

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 easy and natural to use, 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 language features, or non-deterministic ones, or concurrency.

A big-step SOS of a programming language or calculus is given as a formal *proof system* (see Section 2.1.5). 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

$$\frac{C_1 \Downarrow R_1 \quad C_2 \Downarrow R_2 \quad \ldots \quad C_n \Downarrow R_n}{C_0 \Downarrow R_0} \text{ [if } \textit{condition}\text{]}$$

where C_0, C_1, C_2, ..., C_n are configurations holding fragments of program together with all the needed semantic components, where R_0, R_1, R_2, ..., R_n are *result configurations*, or *irreducible configurations*, i.e., configurations which cannot be reduced anymore, and where *condition* is an optional *side condition*; as discussed in Section 2.1.5, 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's addition is

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

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:
$\langle \_, \_ \rangle$: $AExp \times State \to Configuration$
$\langle \_ \rangle$: $Int \to Configuration$
$\langle \_, \_ \rangle$: $BExp \times State \to Configuration$
$\langle \_ \rangle$: $Bool \to Configuration$
$\langle \_, \_ \rangle$: $Stmt \times State \to Configuration$
$\langle \_ \rangle$: $State \to Configuration$
$\langle \_ \rangle$: $Pgm \to Configuration$

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 2.4.6), 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 2.4.6 (see Figure 2.7) 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 derive sequents, or 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[2], 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* 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 2.1.5 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-Int) and (BigStep-Lookup) define the obvious semantics of integers and of variable lookup; these rules have no premises because integers and variables 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 $\mathtt{x}, \mathtt{y} \in \mathit{Id}$):

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

[2]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 20.

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

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

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

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

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

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

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

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

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

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

$$\langle \{\}, \sigma\rangle \Downarrow \langle \sigma\rangle \qquad \text{(BigStep-Empty-Block)}$$

$$\frac{\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \{\ s\ \}, \sigma\rangle \Downarrow \langle \sigma'\rangle} \qquad \text{(BigStep-Block)}$$

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

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

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

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

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

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

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

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

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

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

The rule above is indeed a correct instance of (BigStep-Div), but, however, one will never be able to infer $\langle 2, (\mathtt{x} \mapsto 8, \mathtt{y} \mapsto 0)\rangle \Downarrow \langle 4\rangle$, 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.):

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

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

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

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

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 multi-threaded 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 attempt to do it is to work with *sets* of result configurations instead of just with result configurations and thus associate to each fragment of program in a state the set of all the results that it can non-deterministically yield. However, such an approach would significantly complicate the big-step definition, so we prefer to not do it. Moreover, since IMP has no side effects yet (we will add it side effects in Section 3.5), 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 = `true` and one for t = `false`. The rule (BigStep-Leq) allows to derive Boolean sequents from arithmetic ones; although we want <= to evaluate

its arguments from left to right, there is no need to do anything more at this stage, because expressions have no side effects in IMP; in Section 3.5 we will see how to achieve the desired evaluation strategy in the presence of side effects in expressions. 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-Empty-Block) states that {} does nothing with the state. (BigStep-Block) states that the block has the same semantics as the enclosed statement, which is correct at this stage because IMP has no local variable declarations; this will change in Section 3.5. (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-Pgm) 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.

Definition 20. *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 20 rather means *termination with no error*. To catch

division-by-zero within the semantics, we need to add a special *error* value that a division by zero would evaluate to, and then to propagate it through all the language constructs (see Exercise 56).

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 BigStep(IMP) $\vdash \langle p \rangle \Downarrow \langle \sigma \rangle$ (see Exercise 57).

Since each rule schema comprises a *recursively enumerable* collection of concrete instances and since we have a finite set of rule schemata, we can enumerate all the instances of all these 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 a terminating IMP program p, 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 insightful 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 58). 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 2.2.1 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 Rewrite 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 illustrated and discussed in Section 3.5.) 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 rewrite 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 rewrite logic theory $\mathcal{R}_{\text{BigStep}}$ in such a way that the corresponding derivation relations are step-for-step equivalent, that is, BigStep $\vdash C \Downarrow R$ if and only if $\mathcal{R}_{\text{BigStep}} \vdash \mathcal{R}_{C \Downarrow R}$, 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 BigStep(IMP) and obtain a rewrite logic semantics of IMP that is step-for-step equivalent to the original big-step semantics of IMP. Finally, we show how $\mathcal{R}_{\text{BigStep(IMP)}}$ can be seamlessly defined in Maude, thus yielding an interpreter for IMP essentially for free.

Faithful Embedding of Big-Step SOS into Rewrite 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 2.1.3, we assume that for any parametric term t (which can be a configuration, a condition, etc.), the term $\bar{t}$ is an equivalent algebraic variant of t of appropriate sort. For example, a parametric configuration C is 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 $\overline{C}$ 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, the algebraic variant of a rule side condition is an appropriate term of sort *Bool*.

To have a formal mechanism for performing reasoning within the employed mathematical domains, which is tacitly assumed in the big step semantics (e.g., $3 +_{Int} 5 = 8$, etc.), in particular for formally evaluating side conditions, we assume that the algebraic signature associated to the various syntactic and semantic categories is extended into a background algebraic specification capable of proving precisely all the equalities over ground terms (i.e., terms containing no variables). As discussed in Section 2.4, this assumption is quite reasonable, because any computational domain is isomorphic to an initial algebra over a finite algebraic specification, and that the latter can prove precisely all the ground equational properties of the domain/initial algebra. Consider, for example, the side condition $\sigma(x) \neq \bot$ of the rules (BigStep-Lookup) and (BigStep-Asgn) in Figure 3.7. Its algebraic variant is the term $\sigma(X) \neq \bot$ of *Bool* sort, where σ and X are variables of sorts *State* and *Id*, respectively. We therefore assume that any ground instance of this *Bool* term (obtained for a ground/concrete instance of the variables σ and X) can be proved using the background algebraic theory equal to either `true` (which means that the map is defined in the given variable) or `false`.

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

$$\frac{C_1 \Downarrow R_1 \quad C_2 \Downarrow R_2 \quad \ldots \quad C_n \Downarrow R_n}{C_0 \Downarrow R_0} \text{ [if } condition\text{]}$$

where C_0, C_1, C_2, ..., C_n are configurations holding fragments of program together with all the needed semantic components, R_0, 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 rewrite logic rule

$$(\forall \mathcal{X})\ \overline{C_0} \to \overline{R_0}\ \textbf{ if }\ \overline{C_1} \to \overline{R_1}\ \wedge\ \overline{C_2} \to \overline{R_2}\ \wedge\ \ldots\ \wedge\ \overline{C_n} \to \overline{R_n}\ [\wedge\ \overline{condition}].$$

where $\mathcal{X}$ is the set of parameters, or meta-variables, that occur in the big-step proof rule (schema), now regarded as variables. 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 two reasonable assumptions about big-step semantics: (1) configurations cannot be nested; and (2) result configurations are irreducible.

Theorem 13. ***(Faithful embedding of big-step SOS into rewrite 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 rewrite logic semantic definition obtained from* BigStep *by translating each rule in* BigStep *as above. (Recall from Section 2.5 that* $\to^1$ *is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewrite logic. Also, since C and R are parameter-free—parameters only appear in rules—,* $\overline{C}$ *and* $\overline{R}$ *are ground terms.)*

Proof. Before we proceed with the proof, let us understand how the assumptions about BigStep and the use of $\rightarrow^1$ affect the rewrite logic proof derivations that can be performed with $\mathcal{R}_{\text{BigStep}}$. First, note that the Reflexivity and Transitivity proof rules of rewrite logic (see Section 2.5) will never apply. Second, since the rules in $\mathcal{R}_{\text{BigStep}}$ correspond to rules in BigStep between configurations and there are no other rules in $\mathcal{R}_{\text{BigStep}}$, and since configurations cannot be nested, we conclude that the Congruence rule of rewrite logic will never apply either. Third, since the rules of BigStep add no equations to $\mathcal{R}_{\text{BigStep}}$, the equations of $\mathcal{R}_{\text{BigStep}}$ correspond all to the background algebraic theory used for domain reasoning; thus, the Equality rule of rewrite logic can be used for domain reasoning and for nothing else. Finally, since configurations cannot be nested and since the result configurations of BigStep are irreducible, we conclude that the Replacement rule of rewrite logic can only instantiate rules in $\mathcal{R}_{\text{BigStep}}$ by substituting their variables with terms (which can be ground or not), but it can perform no "inner" concurrent rewrites. Therefore, the capabilities of rewrite logic are significantly crippled by $\mathcal{R}_{\text{BigStep}}$, as the only deductions that $\mathcal{R}_{\text{BigStep}}$ can perform are domain (equational) reasoning and rule instantiation.

Let Σ be the signature of the assumed background algebraic formalization of configurations.

Let us first assume that BigStep $\vdash C \Downarrow R$ and let us prove that $\mathcal{R}_{\text{BigStep}} \vdash \overline{C} \rightarrow^1 \overline{R}$. We do this proof by structural induction on the BigStep proof/derivation tree of $C \Downarrow R$. The last proof step in the tree deriving $C \Downarrow R$ must correspond to some instance of a rule (schema) of the form

$$\frac{C_1 \Downarrow R_1 \quad ... \quad C_m \Downarrow R_m}{C_0 \Downarrow R_0} \text{ if } \mathit{condition}$$

where $m \geq 0$ and where the *condition* may be missing, in which case we just assume it to be *true*. If $\mathcal{X}$ is the set of parameters, or meta-variables, that occur in this proof rule, then the last proof step in the derivation of $C \Downarrow R$ consists of an instance of the parameters in $\mathcal{X}$ that yields corresponding instances $(C, R, C'_1, R'_1, ..., C'_m, R'_m, \mathit{true})$ of, respectively, $(C_0, R_0, C_1, R_1, ..., C_m, R_m, \mathit{condition})$, such that BigStep $\vdash C'_i \Downarrow R'_i$ for all $1 \leq i \leq m$; moreover, domain reasoning is allowed at any moment, in particular to evaluate the instance of *condition* to *true*. By the (structural) induction hypothesis we have that $\mathcal{R}_{\text{BigStep}} \vdash \overline{C'_i} \rightarrow^1 \overline{R'_i}$ for all $1 \leq i \leq m$. More algebraically, the above say that there is a map $\theta : \mathcal{X} \rightarrow T_\Sigma$ such that:

- (domain reasoning) $\mathcal{R}_{\text{BigStep}} \vdash \overline{C} = \theta(\overline{C}_0)$ and $\mathcal{R}_{\text{BigStep}} \vdash \overline{R} = \theta(\overline{R}_0)$;
- (domain reasoning) $\mathcal{R}_{\text{BigStep}} \vdash \overline{C'_i} = \theta(\overline{C}_i)$ and $\mathcal{R}_{\text{BigStep}} \vdash \overline{R'_i} = \theta(\overline{R}_i)$ for all $1 \leq i \leq m$;
- (domain reasoning) $\mathcal{R}_{\text{BigStep}} \vdash \theta(\overline{\mathit{condition}}) = \texttt{true}$;
- (induction hypothesis) $\mathcal{R}_{\text{BigStep}} \vdash \overline{C'_i} \rightarrow^1 \overline{R'_i}$ for all $1 \leq i \leq m$.

Since $\mathcal{R}_{\text{BigStep}}$ contains the rule

$$(\forall \mathcal{X})\ \overline{C_0} \rightarrow \overline{R_0} \ \textbf{if}\ \overline{C_1} \rightarrow \overline{R_1} \ \wedge\ \overline{C_2} \rightarrow \overline{R_2} \ \wedge\ \ldots \ \wedge\ \overline{C_m} \rightarrow \overline{R_m} \ \wedge\ \overline{\mathit{condition}},$$

the Replacement and Equality rules of rewrite logic then give us $\mathcal{R}_{\text{BigStep}} \vdash \overline{C} \rightarrow^1 \overline{R}$.

Let us now assume that $\mathcal{R}_{\text{BigStep}} \vdash \overline{C} \rightarrow^1 \overline{R}$ and prove that BigStep $\vdash C \Downarrow R$. Since C and R are ground configurations and since all the rewrite rules derived in any given rewrite logic proof contain the same set of variables, it follows that all the rewrite rules derived as part of the proof of $\mathcal{R}_{\text{BigStep}} \vdash \overline{C} \rightarrow^1 \overline{R}$ are also ground. We show by structural induction on the rewrite logic derivation tree that any derivable sequent $\mathcal{R}_{\text{BigStep}} \vdash u \rightarrow^1 v$ with u and v ground is of the form $\mathcal{R}_{\text{BigStep}} \vdash \overline{C_u} \rightarrow^1 \overline{R_v}$, where C_u is a configuration and R_v a result configuration in BigStep such that BigStep $\vdash C_u \Downarrow R_v$. There are two cases to analyze:

- The last step used the Equality proof rule of rewrite logic, that is, $\mathcal{R}_{\text{BigStep}} \vdash u = u'$, $\mathcal{R}_{\text{BigStep}} \vdash u' \rightarrow^1 v'$, and $\mathcal{R}_{\text{BigStep}} \vdash v' = v$. Then by the induction hypothesis there is a configuration $C_{u'}$ and a result configuration $R_{v'}$ such that $u' = \overline{C_{u'}}$, $v' = \overline{R_{v'}}$ and $\text{BigStep} \vdash C_{u'} \Downarrow R_{v'}$. Since $\mathcal{R}_{\text{BigStep}} \vdash u = u'$ and $\mathcal{R}_{\text{BigStep}} \vdash v' = v$, and since equational deduction preserves the sorts of the terms proved equal, it follows that u and v are also of configuration sort. Then let C_u and R_v be the configurations corresponding to u and v, respectively, that is, $u = \overline{C_u}$ and $v = \overline{R_v}$. Since domain properties can be tacitly used in big-step derivations, we conclude that $\text{BigStep} \vdash C_u \Downarrow R_v$ and R_v is a result configuration.

- The last step is an instance of the Replacement rule of rewrite logic, with the rewrite rule

$$(\forall X)\ \overline{C_0} \rightarrow \overline{R_0}\ \textbf{if}\ \overline{C_1} \rightarrow \overline{R_1} \wedge \overline{C_2} \rightarrow \overline{R_2} \wedge \ldots \wedge \overline{C_m} \rightarrow \overline{R_m} \wedge \overline{\mathit{condition}}$$

  Then there is some ground substitution $\theta : \mathcal{X} \rightarrow T_\Sigma$ and ground terms $u_1, v_1, \ldots, u_m, v_m$ such that $u = \theta(\overline{C_0})$ and $v = \theta(\overline{R_0})$, $\theta(\overline{\mathit{condition}}) = \texttt{true}$, and $u_i = \theta(\overline{C_i})$ and $v_i = \theta(\overline{R_i})$ for all $1 \leq i \leq m$. By the induction hypothesis, there are configurations $C_{u_1}, \ldots, C_{u_m}$ and result configurations $R_{v_1}, \ldots, R_{v_m}$ such that $u_i = \overline{C_{u_i}}$, $v_i = \overline{R_{v_i}}$, and $\text{BigStep} \vdash C_{u_i} \Downarrow R_{v_i}$ for all $1 \leq i \leq m$. Let C_u and R_v be the configurations corresponding to u and v, respectively, that is, $u = \overline{C_u}$ and $v = \overline{R_v}$. We have thus found instances $(C_u, R_u, C_{u_1}, R_{v_1}, ..., C_{u_m}, R_{v_m}, \mathit{true})$ of $(C_0, R_0, C_1, R_1, ..., C_m, R_m, \mathit{condition})$, such that $\text{BigStep} \vdash C_{u_i} \Downarrow R_{v_i}$ for all $1 \leq i \leq m$. In other words, we constructed an instance of the BigStep rule

$$\frac{C_1 \Downarrow R_1 \quad \ldots \quad C_m \Downarrow R_m}{C_0 \Downarrow R_0}\ \text{if } \mathit{condition}$$

  with derivations for all its premises. Hence, $\text{BigStep} \vdash C \Downarrow R$.

□

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. The irreducibility of the result configurations guarantees that $\mathcal{R}_{\text{BigStep}}$ does not do more rewrite steps than intended, because rewriting is an inherently transitively closed relation, while the big-step relation $\Downarrow$ is not.

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 ($\rightarrow$ instead of $\Downarrow$ and conditional rewrite rules instead of conditional deduction rules). As Theorem 13 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[3] that supports the one-step rewrite relation $\rightarrow^1$ (that appears in Theorem 13). Indeed, rewrite engines aim at high-performance implementations of the general rewrite relation $\rightarrow$, which may even involve parallel rewriting (see Section 2.5 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 $\rightarrow$.

Given any rewrite theory $\mathcal{R}$, a sufficient condition for $\rightarrow^1$ to be the same as $\rightarrow$ in $\mathcal{R}$ is for the right-hand-sides of the rules in $\mathcal{R}$ to generate terms which make any context that contains them unmatchable by any rule

[3]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.

in $\mathcal{R}$. Fortunately, the two assumptions we made about our original BigStep semantics guarantee this property. First, the non-nestedness assumption guarantees that configurations can only appear at the top of a term, so the only contexts that contain configurations are the configurations themselves. Second, the irreducibility of result configurations assumption ensures that the rules in $\mathcal{R}_{\text{BigStep}}$ will never match any result configurations. Therefore, we can conclude the following important

Corollary 3. *Under the same hypotheses and assumptions as in Theorem 13,*

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

Our current BigStep(IMP) semantics verifies our second assumption, since 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 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 $\rightarrow^1$ will still terminate). There are at least two simple ways to ensure the irreducibility of result configurations, and thus make Corollary 3 still hold:

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*.

Regardless of how the desired property $\text{BigStep} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BigStep}} \vdash \overline{C} \rightarrow \overline{R}$ is ensured, note that, unfortunately, $\mathcal{R}_{\text{BigStep}}$ lacks the main strengths of rewrite logic that make it a good formalism for concurrency: in rewrite 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 Rewrite Logic

Figure 3.8 shows the rewrite 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 rewrite logic convention that variables start with upper-case letters. For the state variable, we used $\mathcal{\sigma}$, 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 $\mathcal{\sigma}(X) \neq \bot$ of Boolean sort.

The following corollary of Theorem 13 and Corollary 3 establishes the faithfulness of the representation of the big-step operational semantics of IMP in rewrite logic:

$$
\begin{array}{rll}
\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 + 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 \,/\, 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 \mathtt{<=} 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 \mathtt{!}\ B, \sigma\rangle \to \langle \mathtt{false}\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \mathtt{true}\rangle \\
\langle \mathtt{!}\ B, \sigma\rangle \to \langle \mathtt{true}\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \mathtt{false}\rangle \\
\langle B_1 \mathtt{\ \&\&\ } B_2, \sigma\rangle \to \langle \mathtt{false}\rangle & \textbf{if} & \langle B_1, \sigma\rangle \to \langle \mathtt{false}\rangle \\
\langle B_1 \mathtt{\ \&\&\ } B_2, \sigma\rangle \to \langle T\rangle & \textbf{if} & \langle B_1, \sigma\rangle \to \langle \mathtt{true}\rangle \wedge \langle B_2, \sigma\rangle \to \langle T\rangle \\
\langle \{\}, \sigma\rangle \to \langle \sigma\rangle & & \\
\langle \{\ S\ \}, \sigma\rangle \to \langle \sigma'\rangle & \textbf{if} & \langle S, \sigma\rangle \to \langle \sigma'\rangle \\
\langle X = A\,\mathtt{;}, \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\ 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 \mathtt{if}\ (B)\ S_1\ \mathtt{else}\ S_2, \sigma\rangle \to \langle \sigma_1\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \mathtt{true}\rangle \wedge \langle S_1, \sigma\rangle \to \langle \sigma_1\rangle \\
\langle \mathtt{if}\ (B)\ S_1\ \mathtt{else}\ S_2, \sigma\rangle \to \langle \sigma_2\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \mathtt{false}\rangle \wedge \langle S_2, \sigma\rangle \to \langle \sigma_2\rangle \\
\langle \mathtt{while}\ (B)\ S, \sigma\rangle \to \langle \sigma\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \mathtt{false}\rangle \\
\langle \mathtt{while}\ (B)\ S, \sigma\rangle \to \langle \sigma'\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \mathtt{true}\rangle \wedge \langle \{\ S\ \mathtt{while}\ (B)\ S\ \}, \sigma\rangle \to \langle \sigma'\rangle \\
\langle \mathtt{int}\ Xl\mathtt{;}\ 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 rewrite logic.

Corollary 4. 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 rewrite logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{BIGSTEP(IMP)}}$, so the two are faithfully equivalent.

★ 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: 5218 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 > .
```

```
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 < ! B,Sigma > => < false >
  if < B,Sigma > => <  true > .
 crl < ! B,Sigma > => < true >
  if < B,Sigma > => < false > .
 crl < B1 && B2,Sigma > => < false >
  if < B1,Sigma > => < false > .
 crl < B1 && B2,Sigma > => < T >
  if < B1,Sigma > => < true > /\ < B2,Sigma > => < T > .

  rl < {},Sigma > => < Sigma > .
 crl < {S},Sigma > => < Sigma' >
  if < S,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) S1 else S2,Sigma > => < Sigma1 >
  if < B,Sigma > => < true > /\ < S1,Sigma > => < Sigma1 > .
 crl < if (B) S1 else S2,Sigma > => < Sigma2 >
  if < B,Sigma > => < false > /\ < S2,Sigma > => < Sigma2 > .
 crl < while (B) S,Sigma > => < Sigma >
  if < B,Sigma > => < false > .
 crl < while (B) S,Sigma > => < Sigma' >
  if < B,Sigma > => < true > /\ < S while (B) S,Sigma > => < Sigma' > .

 crl < int 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.

will give the same output as above. Recall from Section 2.5.6 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 13, Maude's implicit search mechanism in conditions effectively achieves a proof searcher for big-step SOS derivations.

Once one has a rewrite 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 it 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 of our type system (each used variable is declared) is context dependent.

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

$$xl \vdash x : \texttt{int} \quad \text{if } x \in xl \qquad \text{(BigStepTypeSystem-Lookup)}$$

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

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

$$xl \vdash t : \texttt{bool} \quad \text{if } t \in \{\texttt{true}, \texttt{false}\} \qquad \text{(BigStepTypeSystem-Bool)}$$

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

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

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

$$xl \vdash \{\} : \texttt{block} \qquad \text{(BigStepTypeSystem-Empty-Block)}$$

$$\frac{xl \vdash s : \tau}{xl \vdash \{\ s\ \} : \texttt{block}} \quad \text{if } \tau \in \{\texttt{block}, \texttt{stmt}\} \qquad \text{(BigStepTypeSystem-Block)}$$

$$\frac{xl \vdash a : \texttt{int}}{xl \vdash x = a\texttt{;} : \texttt{stmt}} \quad \text{if } x \in xl \qquad \text{(BigStepTypeSystem-Asgn)}$$

$$\frac{xl \vdash s_1 : \tau_1 \quad xl \vdash s_2 : \tau_2}{xl \vdash s_1 \ s_2 : \texttt{stmt}} \quad \text{if } \tau_1, \tau_2 \in \{\texttt{block}, \texttt{stmt}\} \qquad \text{(BigStepTypeSystem-Seq)}$$

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

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

$$\frac{xl \vdash s : \tau}{\vdash \texttt{int}\ xl\texttt{;}\ s : \texttt{pgm}} \quad \text{if } \tau \in \{\texttt{block}, \texttt{stmt}\} \qquad \text{(BigStepTypeSystem-Pgm)}$$

Figure 3.10: BigStepTypeSystem(IMP) — Type system of IMP using big-step SOS ($xl \in \textbf{List\{}Id\textbf{\}}$; $i \in Int$; $x \in Id$; $a, a_1, a_2 \in AExp$; $b, b_1, b_2 \in BExp$; $s, s_1, s_2 \in Stmt$).

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 $tree_1$:

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

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 $tree_2$:

$$tree_2 \quad = \quad \left\{ \begin{array}{c} \dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : \texttt{int}} \quad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{0} : \texttt{int}}}{\texttt{n,s} \vdash \texttt{(n <= 0)} : \texttt{bool}} \\ \hline \texttt{n,s} \vdash \texttt{(!(n <= 0))} : \texttt{bool} \end{array} \right.$$

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

$$tree_3 \quad = \quad \left\{ \begin{array}{c} \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{s} : \texttt{int}} \quad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : \texttt{int}}}{\texttt{n,s} \vdash \texttt{(s + n)} : \texttt{int}} \quad \dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : \texttt{int}} \quad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{-1} : \texttt{int}}}{\texttt{n,s} \vdash \texttt{(n + -1)} : \texttt{int}}}{\texttt{n,s} \vdash \texttt{(s = s + n;)} : \texttt{stmt} \quad \texttt{n,s} \vdash \texttt{(n = n + -1;)} : \texttt{stmt}} \\ \hline \texttt{n,s} \vdash \texttt{(s = s + n; n = n + -1;)} : \texttt{stmt} \\ \hline \texttt{n,s} \vdash \texttt{\{ s = s + n; n = n + -1; \}} : \texttt{block} \end{array} \right.$$

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

$$\dfrac{\dfrac{tree_1 \quad \dfrac{tree_2 \quad tree_3}{\texttt{n,s} \vdash \texttt{(while (!(n <= 0)) \{s = s + n; n = n + -1;\})} : \texttt{stmt}}}{\texttt{n,s} \vdash \texttt{(n = 100; s = 0; while (!(n <= 0)) \{s = s + n; n = n + -1;\})} : \texttt{stmt}}}{\vdash \texttt{(int n,s; n = 100; s = 0; while (!(n <= 0)) \{s = s + n; n = n + -1;\})} : \texttt{pgm}}$$

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.

$$
\begin{array}{rl}
\langle I, Xl\rangle \to \langle \texttt{int}\rangle & \\
\langle X, (Xl, X, Xl')\rangle \to \langle \texttt{int}\rangle & \\
\langle A_1 + A_2, Xl\rangle \to \langle \texttt{int}\rangle & \textbf{if } \langle A_1, Xl\rangle \to \langle \texttt{int}\rangle \wedge \langle A_2, Xl\rangle \to \langle \texttt{int}\rangle \\
\langle \div IMP A_1 A_2, Xl\rangle \to \langle \texttt{int}\rangle & \textbf{if } \langle A_1, Xl\rangle \to \langle \texttt{int}\rangle \wedge \langle A_2, Xl\rangle \to \langle \texttt{int}\rangle \\
\langle A_1 \texttt{ <= } A_2, Xl\rangle \to \langle \texttt{bool}\rangle & \textbf{if } \langle A_1, Xl\rangle \to \langle \texttt{int}\rangle \wedge \langle A_2, Xl\rangle \to \langle \texttt{int}\rangle \\
\langle T, Xl\rangle \to \langle \texttt{bool}\rangle & \\
\langle \texttt{! } B, Xl\rangle \to \langle \texttt{bool}\rangle & \textbf{if } \langle B, Xl\rangle \to \langle \texttt{bool}\rangle \\
\langle B_1 \texttt{ \&\& } B_2, Xl\rangle \to \langle \texttt{bool}\rangle & \textbf{if } \langle B_1, Xl\rangle \to \langle \texttt{bool}\rangle \wedge \langle B_2, Xl\rangle \to \langle \texttt{bool}\rangle \\
\langle \{\}, Xl\rangle \to \langle \texttt{block}\rangle & \\
\langle \{\, S\, \}, Xl\rangle \to \langle \texttt{block}\rangle & \textbf{if } \langle S, Xl\rangle \to \langle ST\rangle \\
\langle X = A\,\texttt{;}, (Xl, X, Xl')\rangle \to \langle \texttt{stmt}\rangle & \textbf{if } \langle A, (Xl, X, Xl')\rangle \to \langle \texttt{int}\rangle \\
\langle S_1\ S_2, Xl\rangle \to \langle \texttt{stmt}\rangle & \textbf{if } \langle S_1, Xl\rangle \to \langle ST_1\rangle \wedge \langle S_2, Xl\rangle \to \langle ST_2\rangle \\
\langle \texttt{if (}B\texttt{) } S_1 \texttt{ else } S_2, Xl\rangle \to \langle \texttt{stmt}\rangle & \textbf{if } \langle B, Xl\rangle \to \langle \texttt{bool}\rangle \wedge \langle S_1, Xl\rangle \to \langle \texttt{block}\rangle \wedge \langle S_2, Xl\rangle \to \langle \texttt{block}\rangle \\
\langle \texttt{while (}B\texttt{) } S, Xl\rangle \to \langle \texttt{stmt}\rangle & \textbf{if } \langle B, Xl\rangle \to \langle \texttt{bool}\rangle \wedge \langle S, Xl\rangle \to \langle \texttt{block}\rangle \\
\langle \texttt{int } Xl\texttt{; } S\rangle \to \langle \texttt{pgm}\rangle & \textbf{if } \langle S, Xl\rangle \to \langle ST\rangle
\end{array}
$$

Figure 3.11: $\mathcal{R}_{\text{BigStepTypeSystem(IMP)}}$: type system of IMP using big-step SOS in rewrite logic. Assume a sort *Type* with constants `int`, `bool`, and `pgm`, and a subsort of it *StmtType* with constants `block` and `stmt`. The sorts of the involved variables is understood (from Figure 3.10), except for T which has the sort *Bool* and for ST, ST_1 and ST_2 which have the sort *StmtType*.

Figure 3.11 shows a translation of the big-step SOS in Figure 3.10 into a rewrite logic theory, following the general technique described in Section 3.2.3 but with two simple optimizations explained shortly. This translation is based on the 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`, `block`, `stmt`, or `pgm`.

The two optimizations refer to how to test the various memberships that occur in some of the rules' side conditions. One optimization is to use associative matching to check membership to a list; this applies to the rules (BigStepTypeSystem-Lookup) and (BigStepTypeSystem-Asgn). Another optimization is to use a (sub)sort as an umbrella for a set of terms of interest (constants in our case); for example, we let T be a variable of sort *Bool* (i.e., T can match both `true` and `false`) when defining the rewrite rule corresponding to (BigStepTypeSystem-Bool), and we let ST, ST_1, and ST_2 be variables ranging over a new subsort *StmtType* of *Type* when defining the rewrite rules corresponding to (BigStepTypeSystem-Block), (BigStepTypeSystem-Seq) and (BigStepTypeSystem-Pgm).

By Corollary 3 we have that a program p is well-typed, that is, $\vdash\ p : \texttt{pgm}$ is derivable with the proof system in Figure 3.10, if and only if $\mathcal{R}_{\text{BigStepTypeSystem(IMP)}} \vdash \langle \overline{p}\rangle \to \langle \texttt{pgm}\rangle$.

```
mod IMP-TYPES is
  sorts StmtType Type .  subsort StmtType < Type .
  ops block stmt : -> StmtType .
  ops int bool 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 .
  var ST ST1 ST2 : StmtType .

  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 < ! B,Xl > => < bool >
  if < B,Xl > => < bool > .
 crl < B1 && B2,Xl > => < bool >
  if < B1,Xl > => < bool > /\ < B2,Xl > => < bool > .

  rl < {},Xl > => < block > .
 crl < {S},Xl > => < block >
  if < S,Xl > => < ST > .
 crl < X = A ;,(Xl,X,Xl') > => < stmt >
  if < A,(Xl,X,Xl') > => < int > .
 crl < S1 S2,Xl > => < stmt >
  if < S1,Xl > => < ST1 > /\ < S2,Xl > => < ST2 > .
 crl < if (B) S1 else S2,Xl > => < stmt >
  if < B,Xl > => < bool > /\ < S1,Xl > => < block > /\ < S2,Xl > => < block > .
 crl < while (B) S,Xl > => < stmt >
  if < B,Xl > => < bool > /\ < S,Xl > => < block > .

 crl < int Xl ; S > => < pgm >
  if < S,Xl > => < ST > .
endm
```

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

⋆ 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

```
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: 20 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 2.1.4). 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 2.1.4), 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 [34] 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 [34]). 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 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, deterministic 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. He demonstrated how one can use big-step SOS to define all these in his seminal paper [34], using the Mini-ML language.

As Kahn himself acknowledged, the idea of using proof systems to capture the operational semantics of programming languages goes back to Plotkin [60, 61] 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 (which may help in formal reasoning), 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.* [49]. Several types of big-step sequents were used in [49], 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 [49], 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 [53], 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 [32] (1990) and Winskel [87] (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 [32] 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 [32] 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 [32] 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 [32] and of Milner *et al.* [49] are somehow reminiscent of original Kahn's notation, Winskel [87] uses a completely different notation. Specifically, he prefers to use big-step sequents of the form $\langle a, \sigma \rangle \rightarrow i$ 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 an 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 $C \rightarrow C'$ 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 [84] proposed big-step SOS implementations in Maude for several languages, including Hennessy's languages [32] and Kahn's Mini-ML [34]. 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 rewrite logic that work for any language. Besides Vardejo and Martí-Oliet, several other authors used rewrite 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 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".

There is a common misconception that big-step SOS is "efficient" when executed. This misconception is fueled by case studies where efficient interpreters for various (deterministic or deterministic fragments of) languages, were more or less mechanically derived from the big-step SOS of those languages. Recall that a big-step SOS is a formal proof system, and that proof systems are typically meant to tell *what* is possible in the defined language, calculus or system, and not *how* to implement it. A big-step SOS *can* indeed be used as a basis to develop efficient interpreters, but one should be aware of the fact that when that happens it happens either because the big-step SOS has a particularly convenient form, where at most one proof rule can apply at any given moment[4], or because one cuts corners in the implementation by deliberately[5] ignoring the proof search process needed to detect which proof rule applies, and instead arbitrarily picking one matching rule and stopping the execution if its premises cannot be proved. Section 3.5 illustrates a big-step SOS whose rules are not syntax-driven (e.g., expressions have side effects and arithmetic operators are non-deterministic); as discussed there, its faithful Maude implementation is indeed very slow, requiring exponential complexity in some cases to derive the rule premises.

3.2.6 Exercises

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

Exercise 54. *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 55. *Change the big-step semantics of the* IMP *conjunction so that it is not short-circuited.*

Exercise 56. *Add an "error" state and modify the big-step semantics in Figure 3.7 to allow derivations of sequents of the form $\langle s, \sigma\rangle \Downarrow \langle error\rangle$ or $\langle p\rangle \Downarrow \langle error\rangle$ when s evaluated in state σ or when p evaluated in the initial state performs a division by zero.*

Exercise 57. *Prove that the transition relation defined by the* BigStep(IMP) *proof system in Figure 3.7 is* ***deterministic****, that is:*

- *If* BigStep(IMP) $\vdash \langle a, \sigma\rangle \Downarrow \langle i\rangle$ *and* BigStep(IMP) $\vdash \langle a, \sigma\rangle \Downarrow \langle i'\rangle$ *then* $i = i'$*;*
- *If* BigStep(IMP) $\vdash \langle b, \sigma\rangle \Downarrow \langle t\rangle$ *and* 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* BigStep(IMP) $\vdash \langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$*;*
- *If p terminates then there is a unique σ such that* 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 56.

Exercise 58. *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****.*

[4] We call such big-step SOS definitions *syntax-driven*.

[5] Either because one proved that this is sound, or because one's intention is to trade soundness for performance.

3.3 Small-Step Structural Operational Semantics (Small-Step SOS)

Known also under the names *transition semantics*, *reduction semantics*, *one-step operational semantics*, and *computational semantics*, small-step structural operational semantics, or small-step SOS for short, formally captures the intuitive notion of one atomic computational step. Unlike in big-step SOS where one defines all computation steps in one transition, in a small-step SOS a transition encodes only one computation step. To distinguish small-step from big-step transitions, we use a plain arrow $\rightarrow$ instead of $\Downarrow$. To execute a small-step SOS, or to relate it to a big-step SOS, we need to transitively close the small-step transition relation. Indeed, the conceptual relationship between big-step SOS and small-step SOS is that for any configuration C and any result configuration R, $C \Downarrow R$ if and only if $C \rightarrow^\star R$. Small-step SOS is typically preferred over big-step SOS when defining languages with a high-degree of non-determinism, such as, for example, concurrent languages, because in a small-step semantics one has direct control over what can execute and when.

Like big-step SOS, a small-step SOS of a programming language or calculus is also given as a formal proof system (see Section 2.1.5). The *small-step SOS sequents* are also binary relations over configurations like in big-step SOS, but in small-step SOS they are written $C \rightarrow C'$ and have the meaning that C' is a configuration obtained from C after *one step* of computation. A *small-step SOS rule* therefore has the form

$$\frac{C_1 \rightarrow C_1' \quad C_2 \rightarrow C_2' \quad \ldots \quad C_n \rightarrow C_n'}{C_0 \rightarrow C_0'} \text{ [if } \textit{condition}\text{]}$$

where $C_0, C_0', C_1, C_1', C_2, C_2', \ldots, C_n, C_n'$ are configurations holding fragments of program together with all the needed semantic components, like in big-step SOS, and *condition* is an optional side condition. Unlike in big-step SOS, the result configurations do not need to be explicitly defined. In small-step SOS they are implicit: they are precisely those configurations which cannot be reduced anymore using the one-step relation.

Given a configuration holding a fragment of program, a small-step of computation typically takes place in some subpart of the fragment. However, when each of the subparts is already reduced, then the small-step can apply on the part itself. A small-step SOS is therefore finer-grain than big-step SOS, and thus more verbose, because one has to cover all the cases where a computation step can take place. For example, the small-step SOS of addition in IMP is

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1' + a_2, \sigma\rangle}$$

$$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1 + a_2', \sigma\rangle}$$

$$\langle i_1 + i_2, \sigma\rangle \rightarrow \langle i_1 +_{Int} i_2, \sigma\rangle$$

Here, the meaning of a relation $\langle a, \sigma\rangle \rightarrow \langle a', \sigma\rangle$ is that arithmetic expression a in state σ is reduced, in one small-step, to arithmetic expression a' and the state stays unchanged. Like for big-step SOS, one can encounter various other notations for small-step SOS configurations in the literature, e.g., $[a, \sigma]$, or (a, σ), or $\{a, \sigma\}$, or $\langle a \mid \sigma\rangle$, etc. Like for big-step SOS, we prefer to uniformly use the angle-bracket-and-comma notation $\langle a, \sigma\rangle$. Also, like for big-step SOS, one can encounter various decorations on the transition arrow $\rightarrow$, a notable situation being when the transition is *labeled*. Again like for big-step SOS, we assume that such transition decorations are incorporated in the (source and/or target) configurations. How this can be effectively achieved is discussed in detail in Section 3.6 in the context of modular SOS (which allows rather complex transition labels).

The rules above rely on the fact that expression evaluation in IMP has no side effects. If there were side effects, like in the IMP extension in Section 3.5, then the σ's in the right-hand side configurations above

would need to change to a different symbol, say σ', to account for the possibility that the small-step in the condition of the rules, and implicitly in their conclusion, may change the state as well. While in big-step SOS it is more common to derive sequents of the form $\langle a, \sigma \rangle \Downarrow \langle i \rangle$ than of the form $\langle a, \sigma \rangle \Downarrow \langle i, \sigma \rangle$, in small-step SOS the opposite tends to be norm, that is, it is more common to derive sequents of the form $\langle a, \sigma \rangle \to \langle a', \sigma \rangle$ than of the form $\langle a, \sigma \rangle \to \langle a' \rangle$. Nevertheless, the latter sequent type also works when defining languages like IMP whose expressions are side-effect-free (see Exercise 68). Some language designers may prefer this latter style, to keep sequents minimal. However, even if one prefers these simpler sequents, we still keep the angle brackets in the right-hand sides of the transition relations (for the same reason like in big-step SOS—to maintain a uniform notation); in other words, we write $\langle a, \sigma \rangle \to \langle a' \rangle$ and not $\langle a, \sigma \rangle \to a'$.

In addition to rules, a small-step SOS may also include *structural identities*. For example, we can state that sequential composition is associative using the following structural identity:

$$(s_1 \ s_2) \ s_3 \equiv s_1 \ (s_2 \ s_3)$$

The small-step SOS rules apply *modulo* structural identities. In other words, the structural identities can be used anywhere in any configuration and at any moment during the derivation process, without counting as computational steps. In practice, they are typically used to rearrange the syntactic term so that some small-step SOS rule can apply. In particular, the structural rule above allows the designer of the small-step SOS to rely on the fact that the first statement in a sequential composition is *not* a sequential composition, which may simplify the actual SOS rules; this is indeed the case in Section 3.5.4, where we extend IMP with dynamic threads (we do not need structural identities in the small-step SOS definition of the simple IMP language in this section). Structural identities are not easy to execute and/or implement in their full generality, because they can quickly yield an exponential explosion in the number of terms that need to be matched by rules. Their role in SOS is the same as the role of equations in rewriting logic definitions; in fact, we effectively turn them into equations when we embed small-step SOS into rewrite logic (see Section 3.3.3).

3.3.1 IMP Configurations for Small-Step SOS

The configurations needed for the small-step SOS of IMP are a subset of those needed for its big-step SOS in Section 3.2.1. Indeed, we still need all the two-component configurations containing a fragment of program and a state, but, for the particular small-step SOS style that we follow in this section, we do not need those result configurations of big-step SOS containing only a value or only a state. If one prefers to instead follow the minimalist style as in Exercise 68, then one would also need the other configuration types. Here are all the configuration types needed for the small-step SOS of IMP given in the remainder of this section:

- $\langle a, \sigma \rangle$ grouping arithmetic expressions a and states σ;
- $\langle b, \sigma \rangle$ grouping Boolean expressions b and states σ;
- $\langle s, \sigma \rangle$ grouping statements s and states σ;
- $\langle p \rangle$ holding programs p.

We still need the one-component configuration holding only a program, because we still want to reduce a program in the default initial state (empty) without having to mention the empty state.

sorts:
Configuration
operations:
$\langle \_, \_ \rangle$: $AExp \times State \to Configuration$
$\langle \_, \_ \rangle$: $BExp \times State \to Configuration$
$\langle \_, \_ \rangle$: $Stmt \times State \to Configuration$
$\langle \_ \rangle$: $Pgm \to Configuration$

Figure 3.13: IMP small-step SOS configurations as an algebraic signature.

IMP Small-Step SOS Configurations as an Algebraic Signature

Figure 3.13 shows an algebraic signature defining the IMP configurations above, which is needed for the subsequent small-step operational semantics. We defined this algebraic signature in the same style and following the same assumptions as those for big-step SOS in Section 3.2.1.

3.3.2 The Small-Step SOS Rules of IMP

Figures 3.14 and 3.15 show all the rules in our IMP small-step SOS proof system, the former showing the rules corresponding to expressions, both arithmetic and Boolean, and the latter showing those rules corresponding to statements. The sequents that this proof system derives have the forms $\langle a, \sigma \rangle \to \langle a', \sigma \rangle$, $\langle b, \sigma \rangle \to \langle b', \sigma \rangle$, $\langle s, \sigma \rangle \to \langle s', \sigma' \rangle$, and $\langle p \rangle \to \langle s, \sigma \rangle$, where a ranges over *AExp*, b over *BExp*, s over *Stmt*, p over *Pgm*, and σ and σ' over *State*.

The meaning of $\langle a, \sigma \rangle \to \langle a', \sigma \rangle$ is that given state σ, the arithmetic expression a reduces in one (small) step to the arithmetic expression a' and the state σ stays unchanged. The meaning of $\langle b, \sigma \rangle \to \langle b', \sigma \rangle$ is similar but with Boolean expressions instead of arithmetic expressions. The meaning of $\langle s, \sigma \rangle \to \langle s', \sigma' \rangle$ is that statement s in state σ reduces in one step to statement s' in a potentially modified state σ'. The meaning of $\langle p \rangle \to \langle s, \sigma \rangle$ is that program p reduces in one step to statement s in state σ (as expected, whenever such sequents can be derived, the statement s is the body of p and the state σ initializes to 0 the variables declared by p). The reason for which the state stays unchanged in the sequents corresponding to arithmetic and Boolean expressions is because, as discussed, IMP's expressions currently have no side effects; we will have to change these rules later on in Section 3.5 when we add a variable increment arithmetic expression construct to IMP. A small-step reduction of a statement may or may not change the state, so we use a different symbol in the right-hand side of statement transitions, σ', to cover both cases.

We next discuss each of the small-step SOS rules of IMP in Figures 3.14 and 3.15. Before we start, note that we have no rules for reducing constant (integer or Boolean) expressions to their corresponding values as we had in big-step SOS (i.e., no rules corresponding to (BigStep-Int) and (BigStep-Bool) in Figure 3.7). Indeed, we do not want to have small-step rules of the form $\langle v, \sigma \rangle \to \langle v, \sigma \rangle$ because no one-step reductions are further desired on values v: adding such rules would lead to undesired divergent SOS reductions later on when we consider the transitive closure of the one-step relation $\to$. Recall that the big-step SOS relation $\Downarrow$ captured all the reduction steps at once, including zero steps, and thus it did not need to be transitively closed like the small-step relation $\to$, so evaluating values to themselves was not problematic in big-step SOS.

The rule (SmallStep-Lookup) happens to be almost the same as in big-step SOS; that's because variable lookup is an atomic-step operation both in big-step and in small-step SOS. The rules (SmallStep-Add-Arg1), (SmallStep-Add-Arg2), and (SmallStep-Add) give the small-step semantics of addition, and (SmallStep-Div-Arg1), (SmallStep-Div-Arg2), and (SmallStep-Div) that of division, each covering all the three cases where

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

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \to \langle a_1' + a_2, \sigma\rangle} \qquad \text{(SmallStep-Add-Arg1)}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \to \langle a_1 + a_2', \sigma\rangle} \qquad \text{(SmallStep-Add-Arg2)}$$

$$\langle i_1 + i_2, \sigma\rangle \to \langle i_1 +_{Int} i_2, \sigma\rangle \qquad \text{(SmallStep-Add)}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \to \langle a_1' \ / \ a_2, \sigma\rangle} \qquad \text{(SmallStep-Div-Arg1)}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \to \langle a_1 \ / \ a_2', \sigma\rangle} \qquad \text{(SmallStep-Div-Arg2)}$$

$$\langle i_1 \ / \ i_2, \sigma\rangle \to \langle i_1 \ /_{Int} \ i_2, \sigma\rangle \quad \text{if } i_2 \neq 0 \qquad \text{(SmallStep-Div)}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \ \texttt{<=} \ a_2, \sigma\rangle \to \langle a_1' \ \texttt{<=} \ a_2, \sigma\rangle} \qquad \text{(SmallStep-Leq-Arg1)}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle i_1 \ \texttt{<=} \ a_2, \sigma\rangle \to \langle i_1 \ \texttt{<=} \ a_2', \sigma\rangle} \qquad \text{(SmallStep-Leq-Arg2)}$$

$$\langle i_1 \ \texttt{<=} \ i_2, \sigma\rangle \to \langle i_1 \leq_{Int} i_2, \sigma\rangle \qquad \text{(SmallStep-Leq)}$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \texttt{!}\ b, \sigma\rangle \to \langle \texttt{!}\ b', \sigma\rangle} \qquad \text{(SmallStep-Not-Arg)}$$

$$\langle \texttt{! true}, \sigma\rangle \to \langle \texttt{false}, \sigma\rangle \qquad \text{(SmallStep-Not-True)}$$

$$\langle \texttt{! false}, \sigma\rangle \to \langle \texttt{true}, \sigma\rangle \qquad \text{(SmallStep-Not-False)}$$

$$\frac{\langle b_1, \sigma\rangle \to \langle b_1', \sigma\rangle}{\langle b_1 \ \texttt{\&\&} \ b_2, \sigma\rangle \to \langle b_1' \ \texttt{\&\&} \ b_2, \sigma\rangle} \qquad \text{(SmallStep-And-Arg1)}$$

$$\langle \texttt{false \&\& } b_2, \sigma\rangle \to \langle \texttt{false}, \sigma\rangle \qquad \text{(SmallStep-And-False)}$$

$$\langle \texttt{true \&\& } b_2, \sigma\rangle \to \langle b_2, \sigma\rangle \qquad \text{(SmallStep-And-True)}$$

Figure 3.14: SmallStep(IMP)— Small-step SOS of IMP expressions ($i_1, i_2 \in \mathit{Int}$; $x \in \mathit{Id}$; $a_1, a_1', a_2, a_2' \in \mathit{AExp}$; $b, b', b_1, b_1', b_2 \in \mathit{BExp}$; $\sigma \in \mathit{State}$).

a small-step reduction can take place. The first two cases in each group may apply non-deterministically. Recall from Section 3.2 that big-step SOS was inappropriate for defining the desired non-deterministic evaluation strategies for + and / . Fortunately, that was not a big problem for IMP, because its intended non-deterministic constructs are side-effect free. Therefore, the intended non-deterministic evaluation strategies of these particular language constructs did not affect the overall determinism of the IMP language, thus making its deterministic (see Exercise 57) big-step SOS definition in Section 3.2 acceptable. As expected, the non-deterministic evaluation strategies of + and / , which this time can be appropriately captured within the small-step SOS, will not affect the overall determinism of the IMP language (that is, the reflexive/transitive closure $\rightarrow^\star$ of $\rightarrow$; see Theorem 14). These will start making a difference when we add side effects to expressions in Section 3.5.

The rules (SmallStep-Leq-Arg1), (SmallStep-Leq-Arg2), and (SmallStep-Leq) give the deterministic, sequential small-step SOS of <=. The first rule applies whenever a_1 is not an integer, then the second rule applies when a_1 is an integer but a_2 is not an integer, and finally, when both a_1 and a_2 are integers, the third rule applies. The rules (SmallStep-Not-Arg), (SmallStep-Not-True), and (SmallStep-Not-False) are self-explanatory, while the rules (SmallStep-And-Arg1), (SmallStep-And-False) and (SmallStep-And-True) give the short-circuited semantics of and: indeed, b_2 will not be reduced unless b_1 is first reduced to true.

Before we continue with the remaining small-step SOS rules for statements, let us see an example of a small-step SOS reduction using the rules discussed so far; as in the case of the big-step SOS rules in Section 3.2, recall that the small-step SOS rules are also rule schemas, that is, they are parametric in the (meta-)variables a, a_1, b, s, σ, etc. The following is a correct derivation, where x and y are program variables and σ is any state with $\sigma(\mathtt{x}) = 1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\langle \mathtt{x}, \sigma\rangle \rightarrow \langle 1, \sigma\rangle}}{\langle \mathtt{y} \ / \ \mathtt{x}, \sigma\rangle \rightarrow \langle \mathtt{y} \ / \ 1, \sigma\rangle}}{\langle \mathtt{x} + (\mathtt{y} \ / \ \mathtt{x}), \sigma\rangle \rightarrow \langle \mathtt{x} + (\mathtt{y} \ / \ 1), \sigma\rangle}}{\langle (\mathtt{x} + (\mathtt{y} \ / \ \mathtt{x})) <= \mathtt{x}, \sigma\rangle \rightarrow \langle (\mathtt{x} + (\mathtt{y} \ / \ 1)) <= \mathtt{x}, \sigma\rangle}$$

The above can be regarded as a proof of the fact that replacing the second occurrence of x by 1 is a correct one-step computation of IMP, as defined using the small-step SOS rules discussed so far.

Let us now discuss the small-step SOS rules of statements in Figure 3.15. Unlike the reduction of expressions, a reduction step of a statement may also change the state. Note that the empty block, {}, acts as a value that statements evaluate to. Like for other values (integers and Booleans), there is no rule for the empty block, because such a rule would lead to non-termination. Rule (SmallStep-Block) simply dissolves the block construct and keeps the inner statement. We can afford to do this in IMP only because its blocks currently have no local variable declarations (this will change in IMP++ in Section 3.5). An alternative could be to keep the block and instead advance the inner statement (see Exercise 61). Rule (SmallStep-Asgn-Arg2) reduces the second argument—which is an arithmetic expression—of an assignment statement whenever possible, regardless of whether the assigned variable was declared or not. Exercise 62 proposes an alternative semantics where the arithmetic expression is only reduced when the assigned variable has been declared. When the second argument is already fully reduced (i.e., it is an integer value), the rule (SmallStep-Asgn) reduces the assignment statement to {}, at the same time updating the state accordingly. Therefore, two steps are needed in order to assign an already evaluated expression to a variable: one step to write the variable in the state and modify the assignment to {}, and another step to dissolve the resulting {}. Exercise 63 proposes an alternative semantics where these operations can be done in one step.

The rules (SmallStep-Seq-Arg1) and (SmallStep-Seq-Empty-Block) give the small-step SOS of sequential composition: if the first statement is reducible then reduce it, otherwise, if it is {}, move on in a

$$\langle \{\ s\ \}, \sigma\rangle \to \langle s, \sigma\rangle \qquad \text{(SmallStep-Block)}$$

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma\rangle}{\langle x = a\,;, \sigma\rangle \to \langle x = a'\,;, \sigma\rangle} \qquad \text{(SmallStep-Asgn-Arg2)}$$

$$\langle x = i\,;, \sigma\rangle \to \langle \{\}, \sigma[i/x]\rangle \quad \text{if } \sigma(x) \neq \bot \qquad \text{(SmallStep-Asgn)}$$

$$\frac{\langle s_1, \sigma\rangle \to \langle s_1', \sigma'\rangle}{\langle s_1\ \ s_2, \sigma\rangle \to \langle s_1'\ \ s_2, \sigma'\rangle} \qquad \text{(SmallStep-Seq-Arg1)}$$

$$\langle \{\}\ s_2, \sigma\rangle \to \langle s_2, \sigma\rangle \qquad \text{(SmallStep-Seq-Empty-Block)}$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \texttt{if (}b\texttt{)}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \to \langle \texttt{if (}b'\texttt{)}\ s_1\ \texttt{else}\ s_2, \sigma\rangle} \qquad \text{(SmallStep-If-Arg1)}$$

$$\langle \texttt{if (true)}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \to \langle s_1, \sigma\rangle \qquad \text{(SmallStep-If-True)}$$

$$\langle \texttt{if (false)}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \to \langle s_2, \sigma\rangle \qquad \text{(SmallStep-If-False)}$$

$$\langle \texttt{while (}b\texttt{)}\ s, \sigma\rangle \to \langle \texttt{if (}b\texttt{)}\ \{\ s\ \texttt{while (}b\texttt{)}\ s\ \}\ \texttt{else}\ \{\}, \sigma\rangle \qquad \text{(SmallStep-While)}$$

$$\langle \texttt{int}\ xl\texttt{;}\ s\rangle \to \langle s, xl \mapsto 0\rangle \qquad \text{(SmallStep-Pgm)}$$

Figure 3.15: SmallStep(IMP)— Small-step SOS of IMP statements ($i \in \mathit{Int}$; $x \in \mathit{Id}$; $xl \in \mathbf{List\{}\mathit{Id}\mathbf{\}}$; $a, a' \in \mathit{AExp}$; $b, b' \in \mathit{BExp}$; $s, s_1, s_1', s_2 \in \mathit{Stmt}$; $\sigma, \sigma' \in \mathit{State}$).

$$C \to^* C \qquad \text{(SmallStep-Closure-Stop)}$$

$$\frac{C \to C'', \quad C'' \to^* C'}{C \to^* C'} \qquad \text{(SmallStep-Closure-More)}$$

Figure 3.16: SmallStep(IMP)—Reflexive/transitive closure of the small-step SOS relation, which is the same for any small-step SOS of any language or calculus ($C, C', C'' \in \mathit{Configuration}$).

small-step to the second statement. Another possibility (different from that in Exercise 63) to avoid wasting the computational step generated by reductions to {}like in the paragraph above, is to eliminate the rule (SmallStep-Seq-Empty-Block) and instead to add a rule that allows the reduction of the second statement provided that the first one is terminated. This approach is proposed by Hennessy [32], where he introduces a new sequent for *terminated configurations*, say $C\surd$, and then includes a rule like the following (and no rule like (SmallStep-Seq-Empty-Block)):

$$\frac{\langle s_1, \sigma\rangle\surd \quad \langle s_2, \sigma\rangle \to \langle s_2', \sigma'\rangle}{\langle s_1 \;\; s_2, \sigma\rangle \to \langle s_2', \sigma'\rangle}$$

The three rules for the conditional, namely (SmallStep-If-Arg1), (SmallStep-If-True), and (SmallStep-If-False), are straightforward; note that the two branches are never reduced when the condition can still be reduced. Exercise 65 proposes an alternative semantics for the conditional which wastes no computational step on switching to one of the two branches once the condition is evaluated.

The (SmallStep-While) rule unrolls the loop once; this unrolling semantics seems as natural as it can be, but one should notice that it actually also generates an artificial computational step. Exercise 66 proposes an alternative loop semantics which wastes no computational step.

Finally, (SmallStep-Pgm) gives the semantics of programs by reducing them to their body statement in the expected state formed by initializing all the declared variables to 0. Note, however, that this rule also wastes a computational step; indeed, one may not want the initialization of the state with default values for variables to count as a step. Exercise 67 addresses this problem.

It is worthwhile noting that one has some flexibility in how to give a small-step SOS semantics to a language. The same holds true for almost any language definitional style, not only for SOS.

On Proof Derivations, Evaluation, and Termination

To formally capture the notion of "sequence of transitions", in Figure 3.16 we define the relation of *reflexive/transitive closure* of the small-step SOS transition.

Definition 21. *Given appropriate* IMP *small-step SOS configurations C and C′, the* IMP *small-step SOS sequent* $C \to C'$ *is* ***derivable****, written* SmallStep(IMP) $\vdash C \to C'$*, iff there is some proof tree rooted in* $C \to C'$ *which is derivable using the proof system* SmallStep(IMP) *in Figures 3.14 and 3.15. In this case, we also say that C* ***reduces in one step*** *to C′. Similarly, the many-step sequent* $C \to^\star C'$ *is* ***derivable****, written* SmallStep(IMP) $\vdash C \to^\star C'$*, iff there is some proof tree rooted in* $C \to^\star C'$ *which is derivable using the proof system in Figures 3.14, 3.15, and 3.16. In this case, we also say that C* ***reduces*** *(in zero, one, or more steps) to C′. Configuration R is* ***irreducible*** *iff there is no configuration C such that* SmallStep(IMP) $\vdash R \to C$*, and is a* ***result*** *iff it has one of the forms* $\langle i, \sigma\rangle$*,* $\langle t, \sigma\rangle$*, or* $\langle\{\}, \sigma\rangle$*, where* $i \in \mathit{Int}$*,* $t \in \mathit{Bool}$*, and* $\sigma \in \mathit{State}$*. Finally, configuration C* ***terminates*** *under* SmallStep(IMP) *iff there is no infinite sequence of configurations* $C_0, C_1, \ldots$ *such that* $C_0 = C$ *and* C_i *reduces in one step to* C_{i+1} *for any natural number i.*

Result configurations are irreducible, but there are irreducible configurations which are not necessarily result configurations. For example, the configuration $\langle i/0, \sigma \rangle$ is irreducible but it is not a result. Like for big-step SOS, to catch division-by-zero within the semantics we need to add special error values/states and propagate them through all the language constructs (see Exercise 70).

The syntax of IMP (Section 3.1.1, Figure 3.1) was deliberately ambiguous with regards to sequential composition, and that was motivated by the fact that the semantics of the language will be given in such a way that the syntactic ambiguity will become irrelevant. We can now rigorously prove that is indeed the case, that is, we can prove properties of the like " SmallStep(IMP) $\vdash \langle (s_1\ s_2)\ s_3, \sigma \rangle \rightarrow \langle (s_1'\ s_2)\ s_3, \sigma' \rangle$ if and only if SmallStep(IMP) $\vdash \langle s_1\ (s_2\ s_3), \sigma \rangle \rightarrow \langle s_1'\ (s_2\ s_3), \sigma' \rangle$", etc. Exercise 71 discusses several such properties which, together with the fact that the semantics of no language construct is structurally defined in terms of sequential composition, also says that adding the associativity of sequential composition as a structural identity to the small-step SOS of IMP does not change the set of global behaviors of any IMP program (although we have not added it). However, that will not be the case anymore when we extend IMP with dynamic threads in Section 3.5.4, because the semantics of thread spawning will be given making use of sequential composition in such a way that adding this structural identity will be necessary in order to capture the desired set of behaviors.

Note that there are non-terminating sequences which repeat configurations, as well as non-terminating sequences in which all the configurations are distinct; an example of the former is a sequence generated by reducing the statement `while (true) {}`, while an example of the latter is a sequence generated by reducing `while (!(n <= 0)) { n = n + 1; }`. Nevertheless, in the case of IMP, a configuration terminates if and only if it reduces to some irreducible configuration (see Exercise 72). This is not necessarily the case for non-deterministic languages, such as the IMP++ extension in Section 3.5, because reductions of configurations in such language semantics may non-deterministically choose steps that lead to termination, as well as steps that may not lead to termination. In the case of IMP though, the local non-determinism given by rules like (SmallStep-Add-Arg1) and (SmallStep-Add-Arg2) does not affect the overall determinism of the IMP language (Exercise 73).

Relating Big-Step and Small-Step SOS

As expected, the reflexive/transitive closure $\rightarrow^\star$ of the small-step SOS relation captures the same complete evaluation meaning as the $\Downarrow$ relation in big-step SOS (Section 3.2). Since for demonstrations reasons we deliberately worked with different result configurations in big-step and in small-step SOS, our theorem below looks slightly involved; if we had the same configurations in both semantics, then the theorem below would simply state "for any configuration C and any result configuration R, SmallStep(IMP) $\vdash C \rightarrow^\star R$ if and only if BigStep(IMP) $\vdash C \Downarrow R$":

Theorem 14. *The following equivalences hold for any $a \in AExp$, $i \in Int$, $b \in BExp$, $t \in Bool$, $s \in Stmt$, $p \in Pgm$, and $\sigma, \sigma' \in State$:*

- SmallStep(IMP) $\vdash \langle a, \sigma \rangle \rightarrow^\star \langle i, \sigma' \rangle$ *for a state σ' iff* BigStep(IMP) $\vdash \langle a, \sigma \rangle \Downarrow \langle i \rangle$;
- SmallStep(IMP) $\vdash \langle b, \sigma \rangle \rightarrow^\star \langle t, \sigma' \rangle$ *for a state σ' iff* BigStep(IMP) $\vdash \langle b, \sigma \rangle \Downarrow \langle t \rangle$;
- SmallStep(IMP) $\vdash \langle s, \sigma \rangle \rightarrow^\star \langle \{\}, \sigma' \rangle$ *for a state σ' iff* BigStep(IMP) $\vdash \langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle$;
- SmallStep(IMP) $\vdash \langle p \rangle \rightarrow^\star \langle \{\}, \sigma \rangle$ *for a state σ iff* BigStep(IMP) $\vdash \langle p \rangle \Downarrow \langle \sigma \rangle$.

Note that the small-step SOS relation for IMP is a *recursive*, or *decidable* problem: indeed, given configurations C and C', one can use the small-step proof system in Figures 3.14 and 3.15 to exhaustively check whether indeed $C \to C'$ is derivable or not. Moreover, since the rules for the reflexive/transitive closure relation $\to^\star$ in Figure 3.16 can be used to systematically generate any sequence of reductions, we conclude that the relation $\to^\star$ is *recursively enumerable*. Theorem 14 together with the discussion at the end of Section 3.2.2 and Exercise 58, tell us that $\to^\star$ is properly recursively enumerable, that is, it cannot be recursive. This tells us, in particular, that non-termination of a program p is equivalent to saying that, no matter what the state σ is, SmallStep(IMP) $\vdash \langle p \rangle \to^\star \langle \{\}, \sigma \rangle$ cannot be derived. However, unlike for big-step SOS where nothing else can be said about non-terminating programs, in the case of small-step SOS definitions one can use the small-step relation, $\to$, to observe program executions for any number of steps.

3.3.3 Small-Step SOS in Rewrite Logic

Like for big-step SOS, we can also associate a conditional rewrite rule to each small-step SOS rule and hereby obtain a rewrite logic theory that faithfully (i.e., step-for-step) captures the small-step SOS definition. Additionally, we can associate a rewrite logic equation to each SOS structural identity, because in both cases rules are applied modulo structural identities or equations. An important technical aspect needs to be resolved, though. The rewriting relation of rewrite logic is by its own nature reflexively and transitively closed. On the other hand, the small-step SOS relation is not reflexively and transitively closed by default (its reflexive/transitive closure is typically defined a posteriori, as we did in Figure 3.16). Therefore, we need to devise mechanisms to inhibit rewrite logic's reflexive, transitive and uncontrolled application of rules.

We first show that any small-step SOS, say SmallStep, can be mechanically translated into a rewrite logic theory, say $\mathcal{R}_{\text{SmallStep}}$, in such a way that the corresponding derivation relations are step-for-step equivalent, that is, SmallStep $\vdash C \to C'$ if and only if $\mathcal{R}_{\text{SmallStep}} \vdash \mathcal{R}_{C \to C'}$, where $\mathcal{R}_{C \to C'}$ is the corresponding syntactic translation of the small-step SOS sequent $C \to C'$ into a rewrite logic sequent. Second, we apply our generic translation technique on the small-step SOS formal system SmallStep(IMP) defined in Section 3.3.2 and obtain a rewrite logic semantics of IMP that is step-for-step equivalent to the original small-step SOS of IMP. Finally, we show how $\mathcal{R}_{\text{SmallStep(IMP)}}$ can be seamlessly defined in Maude, thus yielding another interpreter for IMP (in addition to the one similarly obtained from the big-step SOS of IMP in Section 3.2.3).

Faithful Embedding of Small-Step SOS into Rewrite Logic

Like for big-step SOS (Section 3.2.3), to define our translation from small-step SOS to rewrite logic generically, we assume that each parametric configuration C admits an equivalent algebraic variant $\overline{C}$ as 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 (e.g., arithmetic expression $a \in AExp$) gets replaced by a *variable* of corresponding sort in $\overline{C}$ (e.g., variable A of sort $AExp$). Consider now a general-purpose small-step SOS rule of the form

$$\frac{C_1 \to C_1' \quad C_2 \to C_2' \quad \dots \quad C_n \to C_n'}{C_0 \to C_0'} \text{ [if } \textit{condition}\text{]}$$

where C_0, C_0', C_1, C_1', C_2, C_2', …, C_n, C_n' are configurations holding fragments of program together with all the needed semantic components, and *condition* is an optional side condition. Before we introduce our transformation, let us first discuss why the same straightforward transformation that we used in the case of big-step SOS,

$$(\forall \mathcal{X})\ \overline{C_0} \to \overline{C_0'} \ \textbf{if}\ \overline{C_1} \to \overline{C_1'} \wedge \overline{C_2} \to \overline{C_2'} \wedge \dots \wedge \overline{C_n} \to \overline{C_n'} \ [\wedge \overline{\textit{condition}}],$$

where $\mathcal{X}$ is the set of parameters, or meta-variables, that occur in the small-step SOS rule, does not work in the case of small-step SOS. For example, with that transformation, the rewrite rules corresponding to the small-step SOS rules of IMP for assignment (SmallStep-Asgn-Arg2) and (SmallStep-Asgn) in Figure 3.15 would be

$$\langle X = A\,;, \sigma\rangle \to \langle X = A'\,;, \sigma\rangle \quad \textbf{if} \quad \langle A, \sigma\rangle \to \langle A', \sigma\rangle$$
$$\langle X = I\,;, \sigma\rangle \to \langle \{\}, \sigma[I/X]\rangle \quad \textbf{if} \quad \sigma(X) \neq \bot$$

The problem with these rules is that the rewrite of a configuration of the form $\langle \mathtt{x} = i\,;, \sigma\rangle$ for some $\mathtt{x} \in \mathit{Id}$, $i \in \mathit{Int}$ and $\sigma \in \mathit{State}$ may not terminate, applying forever the first rule: in rewrite logic, $\langle i, \sigma\rangle \to \langle i, \sigma\rangle$ because $\to$ is closed under reflexivity. Even if we may somehow solve this reflexivity aspect by defining and then including an additional condition $A \neq A'$, such rules still fail to capture the intended small-step transition, because $\to$ is also closed transitively in rewrite logic, so there could be many small-steps taking place in the condition of the first rule before the rule is applied.

To capture *exactly one step* of reduction, thus avoiding the inherent automatic reflexive and transitive closure of the rewrite relation which is desirable in rewrite logic but not in reduction semantics, we can mark the left-hand side (or, alternatively, the right-hand side) configuration in each rewrite sequent to always be distinct from the other one; then each rewrite sequent comprises precisely one step, from a marked to an unmarked configuration (or vice versa). For example, let us place a $\circ$ in front of all the left-hand side configurations and keep the right-hand side configurations unchanged. Then the generic small-step SOS rule above translates into the rewrite logic rule

$$(\forall \mathcal{X})\ \circ\overline{C_0} \to \overline{C'_0}\ \ \textbf{if}\ \ \circ\overline{C_1} \to \overline{C'_1}\ \wedge\ \circ\overline{C_2} \to \overline{C'_2}\ \wedge\ \ldots\ \wedge\ \circ\overline{C_n} \to \overline{C'_n}\ [\wedge\ \overline{\mathit{condition}}],$$

where $\mathcal{X}$ is the same as above. One can metaphorically think of a marked configuration $\circ\overline{C}$ as a "hot" configuration that needs to be "cooled down" in one step, while of an unmarked configuration $\overline{C}$ as a cool one. Theorem 15 below states as expected that a small-step SOS sequent $C \to C'$ is derivable if and only if the term $\circ\overline{C}$ rewrites in the corresponding rewrite theory to $\overline{C'}$ (which is a normal form). Thus, to enable the resulting rewrite system on a given configuration, one needs to first mark the configuration to be reduced (by placing a $\circ$ in front of it) and then to let it rewrite to its normal form. Since the one-step reduction always terminates, the corresponding rewrite task also terminates.

If the original small-step SOS had structural identities, then we translate them into equations in a straightforward manner: each identity $t \equiv t'$ is translated into an equation $(\forall \mathcal{X})\ \overline{t} = \overline{t'}$, where $\mathcal{X}$ is the set of meta-variables appearing in the structural identity. The only difference between the original structural identity and the resulting equation is that the meta-variables of the former become variables in the latter. The role of the two is the same in their corresponding frameworks and whatever we can do with one in one framework we can equivalently do with the other in the other framework; consequently, to simplify the notation and the presentation, we will make abstraction of structural identities and equations in our theoretical developments in the remainder of this chapter.

To obtain the reflexive and transitive many-step closure of the small-step SOS relation in the resulting rewrite setting and thus to be able to obtain an interpreter for the defined language when executing the rewrite system, we need to devise some mechanism to iteratively apply the one-step reduction step captured by rewriting as explained above. There could be many ways to do that, but one simple and uniform way is to add a new configuration marker, say $\star\overline{C}$, with the meaning that $\overline{C}$ must be iteratively reduced, small-step after small-step, either forever or until an irreducible configuration is reached. Figure 3.17 shows how one can define both configuration markers algebraically (assuming some existing *Configuration* sort, e.g., the one in Figure 3.13). To distinguish the marked configurations from the usual configurations and to also possibly allow several one-step markers at the same time, e.g., $\circ\circ\circ\overline{C}$, which could be useful for debugging/tracing

sorts:
ExtendedConfiguration
subsorts:
Configuration $<$ *ExtendedConfiguration*
operations:
$\circ\_$: *Configuration* $\rightarrow$ *ExtendedConfiguration* // reduce one step
$\star\_$: *Configuration* $\rightarrow$ *ExtendedConfiguration* // reduce all steps
rule:
$\star Cfg \rightarrow \star Cfg'$ **if** $\circ\, Cfg \rightarrow Cfg'$ // where Cfg, Cfg' are variables of sort *Configuration*

Figure 3.17: Representing small-step and many-step SOS reductions in rewrite logic.

reasons, we preferred to define the sort of marked configurations as a supersort of *Configuration*. Note that the rule in Figure 3.17 indeed gives $\star$ its desired reflexive and transitive closure property (the reflexivity follows from the fact that the rewrite relation in rewrite logic is reflexive, so $\star C \rightarrow \star C$ for any configuration term C).

Theorem 15. ***(Faithful embedding of small-step SOS into rewrite logic)*** *For any small-step SOS* SmallStep*, and any* SmallStep *appropriate configurations C and C', the following equivalences hold:*

$$\text{SmallStep} \vdash C \rightarrow C' \iff \mathcal{R}_{\text{SmallStep}} \vdash \circ\, \overline{C} \rightarrow^1 \overline{C'} \iff \mathcal{R}_{\text{SmallStep}} \vdash \circ\, \overline{C} \rightarrow \overline{C'}$$

$$\text{SmallStep} \vdash C \rightarrow^\star C' \iff \mathcal{R}_{\text{SmallStep}} \vdash \star\, \overline{C} \rightarrow \star\, \overline{C'}$$

where $\mathcal{R}_{\text{SmallStep}}$ is the rewrite logic semantic definition obtained from SmallStep *by translating each rule in* SmallStep *as above. (Recall from Section 2.5 that $\rightarrow^1$ is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewrite logic. Also, as C and C' are parameter-free—parameters only appear in rules—, $\overline{C}$ and $\overline{C'}$ are ground terms.)*

Except for transforming parameters into variables, the only apparent difference between SmallStep and $\mathcal{R}_{\text{SmallStep}}$ is that the latter marks (using $\circ$) all the left-hand side configurations and, naturally, uses conditional rewrite rules instead of conditional deduction rules. As Theorem 15 shows, there is a step-for-step correspondence between their corresponding computations (or executions, or derivations). Therefore, similarly to the big-step SOS representation in rewrite logic, the rewrite theory $\mathcal{R}_{\text{SmallStep}}$ *is* the small-step SOS SmallStep, and *not* an encoding of it.

Recall from Section 3.2.3 that in the case of big-step SOS there were some subtle differences between the one-step $\rightarrow^1$ (obtained by dropping the reflexivity and transitivity rules of rewrite logic) and the usual $\rightarrow$ relations in the rewrite theory corresponding to the big-step SOS. The approach followed in this section based on marking configurations, thus keeping the left-hand and the right-hand sides always distinct, eliminates all the differences between the two rewrite relations in the case of the one-step reduction (the two relations are identical on the terms of interest). The second equivalence in Theorem 15 tells us that we can turn the rewrite logic representation of the small-step SOS language definition into an interpreter by simply marking the configuration to be completely reduced with a $\star$ and then letting the rewrite engine do its job.

It is worthwhile noting that like in the case of the big-step SOS representation in rewrite logic, unfortunately, $\mathcal{R}_{\text{SmallStep}}$ lacks the main strengths of rewrite logic: in rewrite logic, rewrite rules can apply under any context and in parallel. Indeed, the rules of $\mathcal{R}_{\text{SmallStep}}$ can only apply at the top, sequentially. This should not surprise because, as stated, $\mathcal{R}_{\text{SmallStep}}$ *is* SmallStep, with all its strengths and limitations. By all means,

both the $\mathcal{R}_{\text{SmallStep}}$ above and the $\mathcal{R}_{\text{BigStep}}$ in Section 3.2.3 are rather poor-style rewrite logic specifications. However, that is normal, because neither big-step SOS nor small-step SOS were meant to have the capabilities of rewrite logic w.r.t. context-insensitivity and parallelism; since their representations in rewrite logic are faithful, one should not expect that they inherit the additional capabilities of rewriting logic (if they did, then the representations would not be step-for-step faithful, so something would be wrong).

Small-Step SOS of IMP in Rewrite Logic

Figure 3.18 gives the rewrite logic theory $\mathcal{R}_{\text{SmallStep(IMP)}}$ that is obtained by applying the procedure above to the small-step SOS of IMP, namely the formal system SmallStep(IMP) presented in Figures 3.14 and 3.15. As usual, we used the rewrite logic convention that variables start with upper-case letters, and like in the rewrite theory corresponding to the big-step SOS of IMP in Figure 3.8, we used σ (a larger σ symbol) for variables of sort *State*. Besides the parameter vs. variable subtle (but not unexpected) aspect, the only perceivable difference between SmallStep(IMP) and $\mathcal{R}_{\text{SmallStep(IMP)}}$ is the different notational conventions they use. The following corollary of Theorem 15 establishes the faithfulness of the representation of the small-step SOS of IMP in rewriting logic:

Corollary 5. SmallStep(IMP) $\vdash C \to C' \iff \mathcal{R}_{\text{SmallStep(IMP)}} \vdash \circ\overline{C} \to \overline{C'}$.

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

★ Maude Definition of IMP Small-Step SOS

Figure 3.19 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{SmallStep(IMP)}}$ in Figure 3.18, including representations of the algebraic signatures of small-step SOS configurations in Figure 3.13 and of their extensions in Figure 3.17, which are needed to capture small-step SOS in rewrite logic. The Maude module `IMP-SEMANTICS-SMALLSTEP` in Figure 3.19 is executable, so Maude, through its rewriting capabilities, yields a small-step SOS interpreter for IMP the same way it yielded a big-step SOS interpreter in Section 3.2.3; 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 also irrelevant, so they were replaced by "..."):

```
rewrites: 7632 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < {},n |-> 0 & s |-> 5050 >
```

Like for the big-step SOS definition in Maude, one can also use any of the general-purpose tools provided by Maude on the small-step SOS definition above. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
search * < sumPgm > =>! Cfg:ExtendedConfiguration .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic. However, a relatively large number of states will be explored, 1709, due to the non-deterministic evaluation strategy of the various language constructs:

```
Solution 1 (state 1708)
states: 1709  rewrites: 9232 in ... cpu (... real) (... rewrites/second)
Cfg:ExtendedConfiguration --> * < {},n |-> 0 & s |-> 5050 >
```

$$\circ\langle X, \sigma\rangle \to \langle \sigma(X), \sigma\rangle \ \textbf{if}\ \sigma(X) \neq \bot$$

$$\circ\langle A_1 + A_2, \sigma\rangle \to \langle A_1' + A_2, \sigma\rangle \ \textbf{if}\ \circ\langle A_1, \sigma\rangle \to \langle A_1', \sigma\rangle$$
$$\circ\langle A_1 + A_2, \sigma\rangle \to \langle A_1 + A_2', \sigma\rangle \ \textbf{if}\ \circ\langle A_2, \sigma\rangle \to \langle A_2', \sigma\rangle$$
$$\circ\langle I_1 + I_2, \sigma\rangle \to \langle I_1 +_{Int} I_2, \sigma\rangle$$

$$\circ\langle A_1 \ / \ A_2, \sigma\rangle \to \langle A_1' \ / \ A_2, \sigma\rangle \ \textbf{if}\ \circ\langle A_1, \sigma\rangle \to \langle A_1', \sigma\rangle$$
$$\circ\langle A_1 \ / \ A_2, \sigma\rangle \to \langle A_1 \ / \ A_2', \sigma\rangle \ \textbf{if}\ \circ\langle A_2, \sigma\rangle \to \langle A_2', \sigma\rangle$$
$$\circ\langle I_1 \ / \ I_2, \sigma\rangle \to \langle I_1 \ /_{Int} \ I_2, \sigma\rangle \ \textbf{if}\ I_2 \neq 0$$

$$\circ\langle A_1 \ \texttt{<=}\ A_2, \sigma\rangle \to \langle A_1' \ \texttt{<=}\ A_2, \sigma\rangle \ \textbf{if}\ \circ\langle A_1, \sigma\rangle \to \langle A_1', \sigma\rangle$$
$$\circ\langle I_1 \ \texttt{<=}\ A_2, \sigma\rangle \to \langle I_1 \ \texttt{<=}\ A_2', \sigma\rangle \ \textbf{if}\ \circ\langle A_2, \sigma\rangle \to \langle A_2', \sigma\rangle$$
$$\circ\langle I_1 \ \texttt{<=}\ I_2, \sigma\rangle \to \langle I_1 \leq_{Int} I_2, \sigma\rangle$$

$$\circ\langle \texttt{!}\ B, \sigma\rangle \to \langle \texttt{!}\ B', \sigma\rangle \ \textbf{if}\ \circ\langle B, \sigma\rangle \to \langle B', \sigma\rangle$$
$$\circ\langle \texttt{! true}, \sigma\rangle \to \langle \texttt{false}, \sigma\rangle$$
$$\circ\langle \texttt{! false}, \sigma\rangle \to \langle \texttt{true}, \sigma\rangle$$

$$\circ\langle B_1 \ \texttt{\&\&}\ B_2, \sigma\rangle \to \langle B_1' \texttt{and} B_2, \sigma\rangle \ \textbf{if}\ \circ\langle B_1, \sigma\rangle \to \langle B_1', \sigma\rangle$$
$$\circ\langle \texttt{false \&\&}\ B_2, \sigma\rangle \to \langle \texttt{false}, \sigma\rangle$$
$$\circ\langle \texttt{true \&\&}\ B_2, \sigma\rangle \to \langle B_2, \sigma\rangle$$

$$\circ\langle \{\ S\ \}, \sigma\rangle \to \langle S, \sigma\rangle$$

$$\circ\langle X = A\,\texttt{;}, \sigma\rangle \to \langle X = A'\,\texttt{;}, \sigma\rangle \ \textbf{if}\ \circ\langle A, \sigma\rangle \to \langle A', \sigma\rangle$$
$$\circ\langle X = I\,\texttt{;}, \sigma\rangle \to \langle \{\}, \sigma[I/X]\rangle \ \textbf{if}\ \sigma(X) \neq \bot$$

$$\circ\langle S_1\ S_2, \sigma\rangle \to \langle S_1'\ S_2, \sigma'\rangle \ \textbf{if}\ \circ\langle S_1, \sigma\rangle \to \langle S_1', \sigma'\rangle$$
$$\circ\langle \{\}\ S_2, \sigma\rangle \to \langle S_2, \sigma\rangle$$

$$\circ\langle \texttt{if (}B\texttt{)}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \to \langle \texttt{if (}B'\texttt{)}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \ \textbf{if}\ \circ\langle B, \sigma\rangle \to \langle B', \sigma\rangle$$
$$\circ\langle \texttt{if (true)}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \to \langle S_1, \sigma\rangle$$
$$\circ\langle \texttt{if (false)}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \to \langle S_2, \sigma\rangle$$

$$\circ\langle \texttt{while (}B\texttt{)}\ S, \sigma\rangle \to \langle \texttt{if (}B\texttt{)}\ \{\ S\ \texttt{while (}B\texttt{)}\ S\ \}\ \texttt{else}\ \{\}, \sigma\rangle$$

$$\circ\langle \texttt{int}\ Xl\texttt{;}\ S\rangle \to \langle S, (Xl \mapsto 0)\rangle$$

Figure 3.18: $\mathcal{R}_{\text{SmallStep(IMP)}}$: the small-step SOS of IMP in rewrite logic.

```
mod IMP-CONFIGURATIONS-SMALLSTEP is including IMP-SYNTAX + STATE .
  sorts Configuration ExtendedConfiguration .
  subsort Configuration < ExtendedConfiguration .
  op <_,_> : AExp State -> Configuration .
  op <_,_> : BExp State -> Configuration .
  op <_,_> : Stmt State -> Configuration .
  op <_> : Pgm -> Configuration .
  op o_ : Configuration -> ExtendedConfiguration [prec 80] .   --- one step
  op *_ : Configuration -> ExtendedConfiguration [prec 80] .   --- all steps
  var Cfg Cfg' : Configuration .
 crl * Cfg => * Cfg' if o Cfg => Cfg' .
endm

mod IMP-SEMANTICS-SMALLSTEP is including IMP-CONFIGURATIONS-SMALLSTEP .
  var X : Id .  var Sigma Sigma' : State .  var I I1 I2 : Int .  var Xl : List{Id} .
  var A A' A1 A1' A2 A2' : AExp .  var B B' B1 B1' B2 : BExp .  var S S' S1 S1' S2 : Stmt .

 crl o < X,Sigma > => < Sigma(X),Sigma > if Sigma(X) =/=Bool undefined .

 crl o < A1 + A2,Sigma > => < A1' + A2,Sigma > if o < A1,Sigma > => < A1',Sigma > .
 crl o < A1 + A2,Sigma > => < A1 + A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
  rl o < I1 + I2,Sigma > => < I1 +Int I2,Sigma > .

 crl o < A1 / A2,Sigma > => < A1' / A2,Sigma > if o < A1,Sigma > => < A1',Sigma > .
 crl o < A1 / A2,Sigma > => < A1 / A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
 crl o < I1 / I2,Sigma > => < I1 /Int I2,Sigma > if I2 =/=Bool 0 .

 crl o < A1 <= A2,Sigma > => < A1' <= A2,Sigma > if o < A1,Sigma > => < A1',Sigma > .
 crl o < I1 <= A2,Sigma > => < I1 <= A2',Sigma > if o < A2,Sigma > => < A2',Sigma > .
  rl o < I1 <= I2,Sigma > => < I1 <=Int I2,Sigma > .

 crl o < ! B,Sigma > => < ! B',Sigma > if o < B,Sigma > => < B',Sigma > .
  rl o < ! true,Sigma > => < false,Sigma > .
  rl o < ! false,Sigma > => < true,Sigma > .

 crl o < B1 && B2,Sigma > => < B1' && B2,Sigma > if o < B1,Sigma > => < B1',Sigma > .
  rl o < false && B2,Sigma > => < false,Sigma > .
  rl o < true && B2,Sigma > => < B2,Sigma > .

  rl o < {S},Sigma > => < S,Sigma > .

 crl o < X = A ;,Sigma > => < X = A' ;,Sigma > if o < A,Sigma > => < A',Sigma > .
 crl o < X = I ;,Sigma > => < {},Sigma[I / X] > if Sigma(X) =/=Bool undefined .

 crl o < S1 S2,Sigma > => < S1' S2,Sigma' > if o < S1,Sigma > => < S1',Sigma' > .
  rl o < {} S2,Sigma > => < S2,Sigma > .

 crl o < if (B) S1 else S2,Sigma > => < if (B') S1 else S2,Sigma >
  if o < B,Sigma > => < B',Sigma  > .
  rl o < if (true) S1 else S2,Sigma > => < S1,Sigma > .
  rl o < if (false) S1 else S2,Sigma > => < S2,Sigma > .

  rl o < while (B) S,Sigma > => < if (B) {S while (B) S} else {},Sigma > .

  rl o < int Xl ; S > => < S,(Xl |-> 0) > .
endm
```

Figure 3.19: The small-step SOS of IMP in Maude, including the definition of configurations.

3.3.4 Notes

Small-step structural operational semantics was introduced as just *structural operational semantics* (SOS; no "small-step" qualifier at that time) by Plotkin in a 1981 technical report (University of Aarhus Technical Report DAIMI FN-19, 1981) that included his lecture notes of a programming language course [60]. For more than 20 years this technical report was cited as the main SOS reference by hundreds of scientists who were looking for mathematical rigor in their programming language research. It was only in 2004 that Plotkin's SOS was finally published in a journal [61].

Small-step SOS is pedagogically discussed in several textbooks, two early notable ones being Hennessy [32] (1990) and Winskel [87] (1993). Hennessy [32] uses the same notation as Plotkin, but Winskel [87] prefers a different one to make it clear that it is a one step semantics: $\langle s, \sigma\rangle \rightarrow_1 \langle s', \sigma'\rangle$. Like for big-step SOS, there is no well-established notation for small-step SOS sequents. There is a plethora of research projects and papers that explicitly or implicitly take SOS as *the* formal language semantics framework. Also, SOS served as a source of inspiration, or of problems to be fixed, to other semantic framework designers, including the author. There is simply too much work on SOS, using it, or modifying it, to attempt to cover it here. We limit ourselves to directly related research focused on capturing SOS as a methodological fragment of rewrite logic.

The marked configuration style that we adopted in this section to faithfully represent small-step SOS in rewrite logic was borrowed from Șerbănuță *et al.* [74]; there, the configuration marker "$\circ$" was called a "configuration modifier". An alternative way to keep the left-hand and the right-hand side configurations distinct was proposed by Meseguer and Braga in [44, 14] in the context of representing MSOS into rewrite logic (see Section 3.6); the idea there was to use two different types of configuration wrappers, one for the left-hand side of the transitions and one for the right-hand side, yielding rewrite logic rules of the form:

$$(\forall \mathcal{X})\ \{\overline{C_0}\} \rightarrow [\overline{C'_0}]\ \ \textbf{if}\ \ \{\overline{C_1}\} \rightarrow [\overline{C'_1}]\ \wedge\ \{\overline{C_2}\} \rightarrow [\overline{C'_2}]\ \wedge \cdots \wedge \{\overline{C_n}\} \rightarrow [\overline{C'_n}]\ [\wedge\ \overline{condition}].$$

The solution proposed by Meseguer and Braga in [44, 14] builds upon experience with a previous representation of MSOS in rewrite logic in [15] as well as with an implementation of it in Maude [13, 16], where the necessity of being able to inhibit the default reflexivity and transitivity of the rewrite relation took shape. We preferred to follow the configuration modifier approach proposed by Șerbănuță *et al.* [74] because it appears to be slightly less intrusive (we only tag the already existing left-hand terms of rules) and more general (the left-hands of rules can have any structure, not only configurations, including no structure at all, as it happens in most of the rules of reduction semantics with evaluation contexts—see Section 3.7, e.g., Figure 3.40).

Vardejo and Martì-Oliet [84] give a Maude implementation of a small-step SOS definition for a simple imperative language similar to our IMP (Hennessy's *WhileL* language [32]), in which they do not attempt to prevent the inherent transitivity of rewriting. While they indeed obtain an executable semantics that is reminiscent of the original small-step SOS of the language, they actually define directly the transitive closure of the small-step SOS relation; they explicitly disable the reflexive closure by checking $C \neq C'$ next to rewrites $C \rightarrow C'$ in rule conditions. A small-step SOS of a simple functional language (Hennessy's *Fpl* language [32]) is also given in [84], following a slightly different style, which avoids the problem above. They successfully inhibit rewriting's inherent transitivity in their definition by using a rather creative rewriting representation style for sequents. More precisely, they work with sequents which appear to the user as having the form $\sigma \vdash a \rightarrow a'$, where σ is a state and a, a' are arithmetic expressions, etc., but they actually are rewrite relations between terms $\sigma \vdash a$ and a' (an appropriate signature to allow that to parse is defined). Indeed, there is no problem with the automatic reflexive/transitive closure of rewriting here because the LHS and the RHS of each rewrite rule have different structures. The simple functional language in [84] was pure (no side effects), so there was no need to include a resulting state in the RHS of their rules; if the language had

side effects, then this Vardejo and Martì-Oliet's representation of small-step SOS sequents in [84] would effectively be the same as the one by Meseguer and Braga in [44, 14] (but using a different notation).

3.3.5 Exercises

Prove the following exercises, all referring to the IMP small-step SOS in Figures 3.14 and 3.15.

Exercise 59. *Change the small-step rules for / so that it short-circuits when a_1 evaluates to* 0*.*

Exercise 60. *Change the small-step SOS of the* IMP *conjunction so that it is not short-circuited.*

Exercise 61. *Change the small-step SOS of blocks so that the block is kept but its inner statement is advanced one step.*

Exercise 62. *One can rightfully argue that the arithmetic expression in an assignment should not be reduced any step when the assigned variable is not declared. Change the small-step SOS of* IMP *to only reduce the arithmetic expression when the assigned variable is declared.*

Exercise 63. *A sophisticated language designer could argue that the reduction of the assignment statement to $emptyBlockIMP$ is an artifact of using small-step SOS, therefore an artificial and undesired step which affects the intended computational granularity of the language. Change the small-step SOS of* IMP *to eliminate this additional small-step.*
Hint: Follow the style in Exercise 68; note, however, that that style will require more rules and more types of configurations, so from that point of view is more complex.

Exercise 64. *Give a proof system for deriving "terminated configuration" sequents $C\surd$.*

Exercise 65. *One could argue that our small-step SOS rules for the conditional waste a computational step when switching to one of the two branches once the condition is evaluated.*

1. *Give an alternative small-step SOS for the conditional which does not require a computational step to switch to one of the two branches.*
2. *Can one do better than that? That is, can one save an additional step by reducing the corresponding branch one step at the same time with reducing the condition to true or false in one step? Hint: one may need terminated configurations, like in Exercise 64.*

Exercise 66. *Give an alternative small-step SOS definition of while loops which wastes no computational step. Hint: do a case analysis on b, like in the rules for the conditional.*

Exercise 67. *Give an alternative small-step SOS definition of variable declarations which wastes no computational steps. Hint: one may need terminated configurations, like in Exercise 64.*

Exercise 68. *Modify the small-step SOS definition of* IMP *such that the configurations in the right-hand sides of the transition sequents are minimal (they should contain both a fragment of program and a state only when absolutely needed). What are the drawbacks of this minimalistic approach, compared to the small-step SOS semantics that we chose to follow?*

Exercise 69. *Show that the small-step SOS resulting from Exercise 68 is equivalent to the one in Figure 3.14 on arithmetic and Boolean expressions, that is, $\langle a, \sigma\rangle \to \langle a', \sigma\rangle$ is derivable with the proof system in Figure 3.14 if and only if $\langle a, \sigma\rangle \to \langle a'\rangle$ is derivable with the proof system in Exercise 68, and similarly for Boolean expressions. However, show that the equivalence does not hold true for statements.*

Exercise 70. *To handle division-by-zero, add "error" values and statements, and modify the small-step SOS in Figures 3.14 and 3.15 to allow derivations of sequents whose right-hand side configurations contain "error" as their syntactic component. See also Exercise 56 (same problem but for big-step SOS).*

Exercise 71. *For any* IMP *statements* s_1, s_1', s_2, s_3 *and any states* σ, σ', *the following hold:*

1. $\textsc{SmallStep}(\text{IMP}) \vdash \langle(\{\}\ s_2)\ s_3, \sigma\rangle \to \langle s_2\ s_3, \sigma\rangle$ *and*
   $\textsc{SmallStep}(\text{IMP}) \vdash \langle\{\}\ (s_2\ s_3), \sigma\rangle \to \langle s_2\ s_3, \sigma\rangle$*; and*

2. $\textsc{SmallStep}(\text{IMP}) \vdash \langle(s_1\ s_2)\ s_3, \sigma\rangle \to \langle(s_1'\ s_2)\ s_3, \sigma'\rangle$ *if and only if*
   $\textsc{SmallStep}(\text{IMP}) \vdash \langle s_1\ (s_2\ s_3), \sigma\rangle \to \langle s_1'\ (s_2\ s_3), \sigma'\rangle$.

Consequently, the following also hold (prove them by structural induction on s_1*):*
$\textsc{SmallStep}(\text{IMP}) \vdash \langle(s_1\ s_2)\ s_3, \sigma\rangle \to^\star \langle s_2\ s_3, \sigma'\rangle$ *if and only if*
$\textsc{SmallStep}(\text{IMP}) \vdash \langle s_1\ (s_2\ s_3), \sigma\rangle \to^\star \langle s_2\ s_3, \sigma'\rangle$ *if and only if*
$\textsc{SmallStep}(\text{IMP}) \vdash \langle s_1, \sigma\rangle \to^\star \langle\{\}, \sigma'\rangle$.

Exercise 72. *With the* $\textsc{SmallStep}(\text{IMP})$ *proof system in Figures 3.14, 3.15, and 3.16, configuration C terminates iff* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star R$ *for some irreducible configuration R.*

Exercise 73. *The small-step SOS of* IMP *is globally deterministic: if* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star R$ *and* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star R'$ *for irreducible configurations R and* R'*, then* $R = R'$*. Show the same result for the proof system detecting division-by-zero as in Exercise 70.*

Exercise 74. *Show that if* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star \langle i, \sigma\rangle$ *for some configuration C, integer i, and state* σ*, then C must be of the form* $\langle a, \sigma\rangle$ *for some arithmetic expression a. Show a similar result for Boolean expressions. For statements, show that if* $\textsc{SmallStep}(\text{IMP}) \vdash C \to^\star \langle\{\}, \sigma\rangle$ *then C must be either of the form* $\langle s, \sigma'\rangle$ *for some statement s and some state* σ'*, or of the form* $\langle p\rangle$ *for some program p.*

Exercise 75. *Prove Theorem 14.*

Exercise 76. *State and show a result similar to Theorem 14 but for the small-step and big-step SOS proof systems in Exercises 70 and 56, respectively.*

3.4 Denotational Semantics

Denotational semantics, also known as *fixed-point semantics*, associates to each syntactically well-defined fragment of program a well-defined, rigorous *mathematical object*. This mathematical object denotes the complete behavior of the fragment of program, no matter in what context it will be used. In other words, the denotation of a fragment of program represents its contribution to the meaning of any program containing it. In particular, equivalence of programs or fragments is immediately translated into equivalence of mathematical objects. The later can be then shown using the entire arsenal of mathematics, which is supposedly better understood and more well-established than that of the relatively much newer field of programming languages. There are no theoretical requirements on the nature of the mathematical domains in which the fragments of program are interpreted, although a particular approach became quite well-established, to an extent that it is by many identified with denotational semantics itself: choose the domains to be appropriate bottomed complete partial orders (abbreviated CPOs; see Section 2.6), and give the denotation of recursive language constructs (including loops, recursive functions, recursive data-structures, recursive types, etc.) as least fixed-points, which exist thanks to Theorem 12.

Each language requires customized denotational semantics, the same way each language required customized big-step or small-step structural operational semantics in the previous sections in this chapter. For the sake of concreteness, below we discuss general denotational semantics notions and notations by means of our running example language, IMP, without attempting to completely define it. Note that the complete denotational semantics of IMP is listed in Figure 3.20 in Section 3.4.1. Consider, for example, arithmetic expressions in IMP (which are side-effect free). Each arithmetic expression can be thought of as the mathematical object which is a partial function taking a state to an integer value, namely the value that the expression evaluates to in the given state. It is a partial function and not a total one because the evaluation of some arithmetic expressions may not be defined in some states, for example due to illegal operations such as division by zero. Thus, we can define the *denotation* of arithmetic expressions as a *total* function

$$[\![\_]\!] : AExp \to (State \rightharpoonup Int)$$

taking arithmetic expressions to *partial* functions from states to integer numbers. As in all the semantics discussed in this chapter, states are themselves partial maps from names to values. In what follows we will follow a common notational simplification and will write $[\![a]\!]\sigma$ instead of $[\![a]\!](\sigma)$ whenever $a \in AExp$ and $\sigma \in State$, and similarly for other syntactic or semantic categories. To avoid ambiguity in the presence of multiple denotation functions, many works on denotational semantics tag the denotation functions with their corresponding syntactic categories, e.g., $AExp[\![\_]\!]$ or $[\![\_]\!]_{AExp}$. Our IMP language is simple enough that we can afford to not add such tags.

Denotation functions are defined inductively, over the structure of the language constructs. For example, if $i \in Int$ then $[\![i]\!]$ is the constant function i, that is, $[\![i]\!]\sigma = i$ for any $\sigma \in State$. Similarly, if $x \in Id$ then $[\![x]\!]\sigma = \sigma(x)$. As it is the case in mathematics, if an undefined value is used to calculate another value, then the resulting value is also undefined. In particular, $[\![x]\!]\sigma$ is undefined when $x \notin Dom(\sigma)$. The denotation of compound constructs is defined in terms of the denotations of the parts. In other words, we say that denotational semantics is *compositional*. For example, $[\![a_1 + a_2]\!]\sigma = [\![a_1]\!]\sigma +_{Int} [\![a_2]\!]\sigma$ for any $a_1, a_2 \in AExp$ and any $\sigma \in State$. For the same reason as above, if any of $[\![a_1]\!]\sigma$ or $[\![a_2]\!]\sigma$ is undefined then $[\![a_1 + a_2]\!]\sigma$ is also implicitly undefined. One can also chose to explicitly keep certain functions undefined in certain states, such as the denotation of division in those states in which the denominator is zero:

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

Note that even though the case where $[\![a_2]\!]\sigma$ is undefined ($\bot$) was not needed to be explicitly listed above (because it falls under the first case, $[\![a_2]\!]\sigma \neq 0$), it is still the case that $[\![a_1 \ / \ a_2]\!]\sigma$ is undefined whenever any of $[\![a_1]\!]\sigma$ or $[\![a_2]\!]\sigma$ is undefined.

An immediate use of denotational semantics is to prove properties about programs. For example, we can show that the addition operation on *AExp* whose denotational semantics was given above is associative. Indeed, we can prove for any $a_1, a_2, a_3 \in AExp$ the following equality of partial functions

$$[\![(a_1 + a_2) + a_3]\!] = [\![a_1 + (a_2 + a_3)]\!]$$

using conventional mathematical reasoning and the fact that the sum $+_{Int}$ in the *Int* domain is associative (see Exercise 77). Note that denotational semantics allows us not only to prove properties about programs or fragments of programs relying on properties of their mathematical domains of interpretation, but also, perhaps even more importantly, it allows us to elegantly formulate such properties. Indeed, what does it mean for a language construct to be associative or, in general, for any desired property over programs or fragments of programs to hold? While one could use any of the operational semantics discussed in this chapter to answer this, denotational semantics gives us one of the most direct means to state and prove program properties.

Each syntactic category is interpreted into its corresponding mathematical domain. For example, the denotations of Boolean expressions and of statements are total functions of the form:

$$[\![\_]\!] : BExp \to (State \rightharpoonup Bool)$$
$$[\![\_]\!] : Stmt \to (State \rightharpoonup State)$$

The former is similar to the one for arithmetic expressions above, so we do not discuss it here. The latter is more interesting. Statements can indeed be regarded as partial functions taking states into resulting states. In addition to partiality due to illegal operations in expressions that statements may involve, such as division by zero, partiality in the denotation of statements may also occur for another important reason: *loops may not terminate*. For example, the statement `while (x <= y) {}` will not terminate in those states in which the value that `x` denotes is less than or equal to that of `y`. Mathematically, we say that the function from states to states that this loop statement denotes is undefined in those states in which the loop statement does not terminate. This will be elaborated shortly, after we discuss other statement constructs.

Since {} does not change the state, its denotation is the identity function, i.e., $[\![\{\}]\!] = 1_{State}$. The assignment statement updates the given state when defined in the assigned variable, that is, $[\![\texttt{x = a;}]\!]\sigma = \sigma[[\![a]\!]\sigma/x]$ when $\sigma(x) \neq \bot$ and $[\![a]\!]\sigma \neq \bot$, and $[\![\texttt{x = a;}]\!]\sigma = \bot$ otherwise. Sequential composition accumulates the state changes of the denotations of the composed statements, so it is precisely the mathematical composition of the corresponding partial functions: $[\![s_1 \ s_2]\!] = [\![s_2]\!] \circ [\![s_1]\!]$.

As an example, let us calculate the denotation of the statement "`x = 1; y = 2; x = 3;`" when `x` $\neq$ `y`, i.e., the function $[\![\texttt{x = 1; y = 2; x = 3;}]\!]$. Applying the denotation of sequential composition twice, we obtain $[\![\texttt{x = 3;}]\!] \circ [\![\texttt{y = 2;}]\!] \circ [\![\texttt{x = 1;}]\!]$. Applying this composed function on a state σ, one gets $([\![\texttt{x = 3;}]\!] \circ [\![\texttt{y = 2;}]\!] \circ [\![\texttt{x = 1;}]\!])\sigma$ equals $\sigma[1/\texttt{x}][2/\texttt{y}][3/\texttt{x}]$ when $\sigma(\texttt{x})$ and $\sigma(\texttt{y})$ are both defined, and equals $\bot$ when any of $\sigma(\texttt{x})$ or $\sigma(\texttt{y})$ is undefined; let σ' denote $\sigma[1/\texttt{x}][2/\texttt{y}][3/\texttt{x}]$. By the definition of function update, one can easily see that σ' can be defined as

$$\sigma'(z) = \begin{cases} 3 & \text{if } z = \texttt{x} \\ 2 & \text{if } z = \texttt{y} \\ \sigma(z) & \text{otherwise,} \end{cases}$$

which is nothing but $\sigma[2/\texttt{y}][3/\texttt{x}]$. We can therefore conclude that the statements "`x = 1; y = 2; x = 3;`" and "`y = 2; x = 3;`" are equivalent, because they have the same denotation.

The denotation of a conditional statement `if (`b`)` s_1 `else` s_2 in a state σ is either the denotation of s_1 in σ or that of s_2 in σ, depending upon the denotation of b in σ:

$$[\![\texttt{if (}b\texttt{)}\ s_1\ \texttt{else}\ s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if} \quad [\![b]\!]\sigma = \texttt{true} \\ [\![s_2]\!]\sigma & \text{if} \quad [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if} \quad [\![b]\!]\sigma = \bot \end{cases}$$

The third case above was necessary, because the first two cases do not cover the entire space of possibilities and, in such situations, one may (wrongly in our context here) understand that the function is underspecified in the remaining cases rather than undefined. Using the denotation of the conditional statement above and conventional mathematical reasoning, we can show, for example, that $[\![\texttt{if (y <= z) \{ x = 1; \} else \{ x = 2; \} x = 3;}]\!]$ is the function taking states σ defined in `x`, `y` and `z` to $\sigma[3/\texttt{x}]$.

The language constructs which admit non-trivial and interesting denotational semantics tend to be those which have a recursive nature. One of the simplest such constructs, and the only one we discuss here (see Section 4.8 for other recursive constructs), is IMP's `while` looping construct. Thus, the question we address next is how to define the denotation functions of the form

$$[\![\texttt{while (}b\texttt{)}\ s]\!] : \mathit{State} \rightharpoonup \mathit{State}$$

where $b \in \mathit{BExp}$ and $s \in \mathit{Stmt}$. What we want is $[\![\texttt{while (}b\texttt{)}\ s]\!]\sigma = \sigma'$ iff the while loop correctly terminates in state σ' when executed in state σ. Such a σ' may not always exist for two reasons:

1. Because b or s is undefined (e.g., due to illegal operations) in σ or in other states encountered during the loop execution; or

2. Because s (which may contain nested loops) or the while loop itself does not terminate.

If w is the partial function $[\![\texttt{while (}b\texttt{)}\ s]\!]$, then its most natural definition would appear to be:

$$w(\sigma) = \begin{cases} w([\![s]\!]\sigma) & \text{if} \quad [\![b]\!]\sigma = \texttt{true} \\ \sigma & \text{if} \quad [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if} \quad [\![b]\!]\sigma = \bot \end{cases}$$

Mathematically speaking, this is a problematic definition for several reasons:

1. The partial function w is defined in terms of itself;

2. It is not clear that such a w exists; and

3. In case it exists, it is not clear that such a w is unique.

To see how easily one can yield inappropriate recursive definitions of functions, we refer the reader to the discussion immediately following Theorem 12, which shows examples of recursive definitions which admit no solutions or which are ambiguous.

We next develop the mathematical machinery needed to rigorously define and reason about partial functions like the w above. More precisely, we frame the mathematics needed here as an instance of the general setting and results discussed in Section 2.6. We strongly encourage the reader to familiarize herself with the definitions and results in Section 2.6 before continuing.

A convenient interpretation of partial functions that may ease the understanding of the subsequent material is as *information* or *knowledge bearers*. More precisely, a partial function $\alpha : \mathit{State} \rightharpoonup \mathit{State}$ can be thought

of as carrying knowledge about some states in *State*, namely exactly those on which α is defined. For such a state $\sigma \in State$, the knowledge that α carries is $\alpha(\sigma)$. If α is not defined in a state $\sigma \in State$ then we can think of it as "α *does not have any information about* σ".

Recall from Section 2.6 that the set of partial functions between any two sets can be organized as a bottomed complete partial order (CPO). In our case, if $\alpha, \beta : State \rightharpoonup State$ then we say that α *is less informative than or as informative as* β, written $\alpha \preceq \beta$, if and only if for any $\sigma \in State$, it is either the case that $\alpha(\sigma)$ is not defined, or both $\alpha(\sigma)$ and $\beta(\sigma)$ are defined and $\alpha(\sigma) = \beta(\sigma)$. If $\alpha \preceq \beta$ then we may also say that β *refines* α or that β *extends* α. Then $(State \rightharpoonup State, \preceq \bot)$ is a CPO, where $\bot : State \rightharpoonup State$ is the partial function which is undefined everywhere.

One can think of each possible iteration of a while loop as an opportunity to refine the knowledge about its denotation. Before the Boolean expression b of the loop `while` b `do` s is evaluated the first time, the knowledge that one has about its denotation function w is the empty partial function $\bot : State \rightharpoonup State$, say w_0. Therefore, w_0 corresponds to no information.

Now suppose that we evaluate the Boolean expression b in some state σ and that it is false. Then the denotation of the while loop should return σ, which suggests that we can refine our knowledge about w from w_0 to the partial function $w_1 : State \rightharpoonup State$, which is an identity on all those states $\sigma \in State$ for which $[\![b]\!]\sigma =$ `false` and which remains undefined in any other state.

So far we have not considered any state in which the loop needs to evaluate its body. Suppose now that for some state σ, it is the case that $[\![b]\!]\sigma =$ `true`, $[\![s]\!]\sigma = \sigma'$, and $[\![b]\!]\sigma' =$ `false`, that is, that the while loop terminates in one iteration. Then we can extend w_1 to a partial function $w_2 : State \rightharpoonup State$, which, in addition to being an identity on those states on which w_1 is defined, that is $w_1 \preceq w_2$, takes each σ as above to $w_2(\sigma) = \sigma'$.

By iterating this process, one can define a partial function $w_k : State \rightharpoonup State$ for any natural number k, which is defined on all those states on which the while loop terminates in *at most* k evaluations of its Boolean condition (i.e., $k - 1$ executions of its body). An immediate property of the partial functions w_0, w_1, w_2, $\ldots$, w_k is that they increasingly refine each other, that is, $w_0 \preceq w_1 \preceq w_2 \preceq \cdots \preceq w_k$. Informally, the partial functions w_k approximate w as k increases; more precisely, for any $\sigma \in State$, if $w(\sigma) = \sigma'$, that is, if the while loop terminates and σ' is the resulting state, then there is some k such that $w_k(\sigma) = \sigma'$. Moreover, $w_n(\sigma) = \sigma'$ for any $n \geq k$.

But the main question still remains unanswered: how to define the denotation $w : State \rightharpoonup State$ of the looping statement `while (`b`)` s? According to the intuitions above, w should be some sort of *limit* of the (infinite) sequence of partial functions $w_0 \preceq w_1 \preceq w_2 \preceq \cdots \preceq w_k \preceq \cdots$. We next formalize all the intuitions above. Let us define the total function

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

taking partial functions $\alpha : State \rightharpoonup State$ to partial functions $\mathcal{F}(\alpha) : State \rightharpoonup State$ as follows:

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

The partial functions w_k defined informally above can be now rigorously defined as $\mathcal{F}^k(\bot)$, where $\mathcal{F}^k$ stays for k compositions of $\mathcal{F}$, and $\mathcal{F}^0$ is by convention the identity function, i.e., $1_{(State \rightharpoonup State)}$ (which is total). Indeed, one can show by induction on k the following property, where $[\![s]\!]^i$ stays for i compositions of

$[\![s]\!] : \mathit{State} \rightharpoonup \mathit{State}$ and $[\![s]\!]^0$ is by convention the identity (total) function on *State*:

$$\mathcal{F}^k(\bot)(\sigma) = \begin{cases} [\![s]\!]^i\sigma & \text{if there is } 0 \leq i < k \text{ s.t. } [\![b]\!][\![s]\!]^i\sigma = \texttt{false} \\ & \text{and } [\![b]\!][\![s]\!]^j\sigma = \texttt{true} \text{ for all } 0 \leq j < i \\ \bot & \text{otherwise} \end{cases}$$

We can also show that the following is a chain of partial functions (Exercise 81 requires the reader to prove, for the IMP language, all these facts mentioned above and below)

$$\bot \preceq \mathcal{F}(\bot) \preceq \mathcal{F}^2(\bot) \preceq \cdots \preceq \mathcal{F}^n(\bot) \preceq \cdots$$

in the CPO $(\mathit{State} \rightharpoonup \mathit{State}, \preceq, \bot)$. As intuitively discussed above, this chain incrementally approximates the desired denotation of `while (`b`)` s. The final step is to realize that $\mathcal{F}$ is a continuous function and thus satisfies the hypotheses of the fixed-point Theorem 12, so we can conclude that the least upper bound (LUB) of the chain above, which by Theorem 12 is the least fixed-point $\mathit{fix}(\mathcal{F})$ of $\mathcal{F}$, is the desired denotation of the while loop, that is,

$$[\![\texttt{while (}b\texttt{) } s]\!] = \mathit{fix}(\mathcal{F})$$

Remarks. First, note that we indeed want the *least* fixed-point of $\mathcal{F}$, and not some arbitrary fixed-point of $\mathcal{F}$, to be the denotation of the while statement. Indeed, any other fixed-points would define states in which the while loop is intended to be undefined. To be more concrete, consider the simple IMP while loop "`while (! (k <= 10)) k = k + 1;`" whose denotation is defined only on those states σ with $\sigma(k) \leq 10$ and, on those states, it is the identity. That is,

$$[\![\texttt{while (! (k <= 10)) k = k + 1;}]\!](\sigma) = \begin{cases} \sigma & \text{if } \sigma(k) \leq 10 \\ \bot & \text{otherwise} \end{cases}$$

Consider now another fixed-point $\gamma : \mathit{State} \rightharpoonup \mathit{State}$ of its corresponding $\mathcal{F}$. While γ must still be the identity on those states σ with $\sigma(k) \leq 10$ (indeed, $\gamma(\sigma) = \mathcal{F}(\gamma)(\sigma) = \sigma$ for any such $\sigma \in \mathit{State}$), it is not enforced to be undefined on any other states. In fact, it can be shown (see Exercise 82) that the fixed-points of $\mathcal{F}$ are precisely those γ as above with the additional property that $\gamma(\sigma) = \gamma(\sigma')$ for any $\sigma, \sigma' \in \mathit{State}$ with $\sigma(k) > 10$, $\sigma'(k) > 10$, and $\sigma(x) = \sigma'(x)$ for any $x \neq k$. Such a γ can be, for example, the following:

$$\gamma(\sigma) = \begin{cases} \sigma & \text{if } \sigma(k) \leq 10 \\ \iota & \text{otherwise} \end{cases}$$

where $\iota \in \mathit{State}$ is some arbitrary but fixed state. Such fixed-points are too informative for our purpose here, since we want the denotation of while to be undefined in all states in which the loop does not terminate. Any other fixed-point of $\mathcal{F}$ which is strictly more informative than $\mathit{fix}(\mathcal{F})$ is simply too informative.

Second, note that the chain $\bot \preceq \mathcal{F}(\bot) \preceq \mathcal{F}^2(\bot) \preceq \cdots \preceq \mathcal{F}^n(\bot) \preceq \cdots$ can be stationary in some cases, but in general it is not. For example, when the loop is well-defined and terminates in any state in some fixed maximum number of iterations which does not depend on the state, its denotation is the (total) function in which the chain stabilizes (which in that case is its LUB and, by Theorem 12, the fixed-point of $\mathcal{F}$). For example, the chain corresponding to the loop "`while (1 <= k && k <= 10) k = k + 1;`" stabilizes in 12 steps, each step adding more states to the domain of the corresponding partial function until nothing can be added anymore: at step 1 all states σ with $\sigma(\texttt{k}) > 10$ or $\sigma(\texttt{k}) < 1$, at step 2 those with $\sigma(\texttt{k}) = 10$, at step 3 those with $\sigma(\texttt{k}) = 9$, ..., at step 11 those with $\sigma(\texttt{k}) = 1$; then no other state is added at step 12, that is, $\mathcal{F}^{12}(\bot) = \mathcal{F}^{11}(\bot)$. However, the chain associated to a loop is not stationary in general. For example, "`while (k <= 0) k = k + 1;`" terminates in any state, but there is no bound on the number of iterations. Consequently, there is no n such that $\mathcal{F}^n(\bot) = \mathcal{F}^{n+1}(\bot)$. Indeed, the later has strictly more information than the former: $\mathcal{F}^{n+1}$ is defined on all those states σ with $\sigma(\texttt{k}) = -n$, while $\mathcal{F}^n$ is not.

3.4.1 The Denotational Semantics of IMP

Figure 3.20 shows the complete denotational semantics of IMP. There is not much to comment on the denotational semantics of the various IMP language constructs, because they have already been discussed above. Note though that the denotation of conjunction captures the desired short-circuited semantics, in that the second conjunct is evaluated only when the first evaluates to `true`. Also, note that the denotation of programs is still a total function for uniformity (in spite of the fact that some programs may not be well-defined or may not terminate), but one into the CPO $State_\perp$ (see Section 2.6); thus, the denotation of a program which is not well-defined is $\perp$. Finally, note that, like in the big-step SOS of IMP in Section 3.2.2, we ignore the non-deterministic evaluation strategies of the `+` and `/` arithmetic expression constructs. In fact, since the denotations of the various language constructs are *functions*, non-deterministic constructs cannot be handled in denotational semantics the same way they were handled in operational semantics. There are ways to deal with non-determinism and concurrency in denotational semantics as discussed at the end of this section, but those are more complex and lead to inefficient interpreters when executed, so we do not consider them in this book. We here limit ourselves to denotational semantics of deterministic languages.

3.4.2 Denotational Semantics in Equational/Rewrite Logic

In order to formalize and execute denotational semantics one needs to formalize and execute the fragment of mathematics that is used by the denotation functions. The size of the needed fragment of mathematics is arbitrary and is typically determined by the particular programming language in question. Since a denotational semantics associates to each program or fragment of program a mathematical object expressed using the formalized language of the corresponding mathematical domain, the faithfulness of any representation/encoding/implementation of denotational semantics into any formalism directly depends upon the faithfulness of the formalizations of the mathematical domains.

The faithfulness of formalizations of mathematical domains is, however, not trivial to characterize in general. The formalization of each mathematical domain requires its own proofs of correctness, and in order for such proofs to make sense we need an alternative, trusted definition of the domain. Consider, for example, the basic domain of natural numbers. One may choose to formalize it using, e.g., Peano-style equational axioms or λ-calculus (see Section 4.5); nevertheless, each of these formalizations needs to be shown correct w.r.t. the mathematical domain of natural numbers, and none of these formalizations is powerful enough to mechanically derive all properties over natural numbers. It is therefore customary when formalizing denotational semantics to simply assume that the formalizations of the mathematical domains are correct; or put differently, to separate the problem of verifying the mathematical domains themselves from the problem of giving a language a denotational semantics using those domains. We here do the same thing: we assume that our equational/rewrite logic formalizations of mathematical domains in Section 2.6.5, which allow us in particular to define and execute higher-order functions and fixed-points for them, are correct. This automatically implies that our equational/rewrite logic representation of denotational semantics is faithful to the original denotational semantics; in other words, unlike for other semantic approaches, no faithfulness theorems for our representation of denotational semantics in equational/rewrite logic are needed.

Denotational Semantics of IMP in Equational/Rewrite Logic

Figure 3.21 shows a direct representation of the denotational semantics of IMP in Figure 3.20 using the mathematical domains formalized in equational/rewrite logic in Section 2.6.5.

To reduce the number of denotation functions defined, we chose to collapse all the syntactic categories under only one sort, *Syntax*. Similarly, to reuse existing operations of the CPO domain in Section 2.6.5 (e.g.,

Arithmetic expression constructs

$[\![\_]\!] : AExp \to (State \rightharpoonup Int)$

$[\![i]\!]\sigma = i$

$[\![x]\!]\sigma = \sigma(x)$

$[\![a_1 + a_2]\!]\sigma = [\![a_1]\!]\sigma +_{Int} [\![a_2]\!]\sigma$

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

Boolean expression constructs

$[\![\_]\!] : BExp \to (State \rightharpoonup Bool)$

$[\![t]\!]\sigma = t$

$[\![a_1 <= a_2]\!]\sigma = [\![a_1]\!]\sigma \leq_{Int} [\![a_2]\!]\sigma$

$[\![\,!\, b]\!]\sigma = \neg_{Bool}([\![b]\!]\sigma)$

$$[\![b_1 \ \&\& \ b_2]\!]\sigma = \begin{cases} [\![b_2]\!]\sigma & \text{if} \quad [\![b_1]\!]\sigma = \texttt{true} \\ \texttt{false} & \text{if} \quad [\![b_1]\!]\sigma = \texttt{false} \\ \bot & \text{if} \quad [\![b_1]\!]\sigma = \bot \end{cases}$$

Statement constructs

$[\![\_]\!] : Stmt \to (State \rightharpoonup State)$

$[\![\{\}]\!]\sigma = \sigma$

$[\![\{\ s\ \}]\!]\sigma = [\![s]\!]\sigma$

$$[\![x = a;]\!]\sigma = \begin{cases} \sigma[[\![a]\!]\sigma/x] & \text{if} \quad \sigma(x) \neq \bot \\ \bot & \text{if} \quad \sigma(x) = \bot \end{cases}$$

$[\![s_1 \ \ s_2]\!]\sigma = [\![s_2]\!][\![s_1]\!]\sigma$

$$[\![\texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if} \quad [\![b]\!]\sigma = \texttt{true} \\ [\![s_2]\!]\sigma & \text{if} \quad [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if} \quad [\![b]\!]\sigma = \bot \end{cases}$$

$[\![\texttt{while}\ (b)\ s]\!] = fix(\mathcal{F})$, where $\mathcal{F} : (State \rightharpoonup State) \to (State \rightharpoonup State)$ defined as

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

Programs

$[\![\_]\!] : Pgm \to State_\bot$

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

Figure 3.20: DENOT(IMP): The denotational semantics of IMP.

sort:

Syntax // generic sort for syntax

subsorts:

$\mathit{AExp}, \mathit{BExp}, \mathit{Stmt}, \mathit{Pgm} < \mathit{Syntax}$ // syntactic categories fall under *Syntax*

$\mathit{Int}, \mathit{Bool}, \mathit{State} < \mathit{CPO}$ // basic domains are regarded as CPOs

operation:

$[\![\_]\!] : \mathit{Syntax} \rightarrow \mathit{CPO}$ // denotation of syntax

equations:

// Arithmetic expression constructs:

$[\![I]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ I$

$[\![X]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \sigma(X)$

$[\![A_1 + A_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{app}_{CPO}([\![A_1]\!], \sigma) +_{Int} \mathtt{app}_{CPO}([\![A_2]\!], \sigma)$

$[\![A_1\ /\ A_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{if}_{CPO}(\mathtt{app}_{CPO}([\![A_2]\!], \sigma) \neq_{Int} 0, \mathtt{app}_{CPO}([\![A_1]\!], \sigma) /_{Int} \mathtt{app}_{CPO}([\![A_2]\!], \sigma), \bot)$

// Boolean expression constructs:

$[\![T]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ T$

$[\![A_1 \mathtt{<=} A_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{app}_{CPO}([\![A_1]\!], \sigma) \leq_{Int} \mathtt{app}_{CPO}([\![A_2]\!], \sigma)$

$[\![\mathtt{!}\ B]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \neg_{Bool}\mathtt{app}_{CPO}([\![B]\!], \sigma)$

$[\![B_1\ \mathtt{\&\&}\ B_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{if}_{CPO}(\mathtt{app}_{CPO}([\![B_1]\!], \sigma), \mathtt{app}_{CPO}([\![B_2]\!], \sigma), \mathtt{false})$

// Statement constructs:

$[\![\{\}]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \sigma$

$[\![\{\ S\ \}]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{app}_{CPO}([\![S]\!], \sigma)$

$[\![X = A\ \mathtt{;}]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{app}_{CPO}(\mathtt{fun}_{CPO}\ \mathit{arg}\ \mathtt{->}\ \mathtt{if}_{CPO}(\sigma(X) \neq_{Int} \bot, \sigma[\mathit{arg}/X], \bot), \mathtt{app}_{CPO}([\![A]\!], \sigma))$

$[\![S_1\ S_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{app}_{CPO}([\![S_2]\!], \mathtt{app}_{CPO}([\![S_1]\!], \sigma))$

$[\![\mathtt{if}\ \mathtt{(}B\mathtt{)}\ S_1\ \mathtt{else}\ S_2]\!] = \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{if}_{CPO}(\mathtt{app}_{CPO}([\![B]\!], \sigma), \mathtt{app}_{CPO}([\![S_1]\!], \sigma), \mathtt{app}_{CPO}([\![S_2]\!], \sigma))$

$[\![\mathtt{while}\ \mathtt{(}B\mathtt{)}\ S]\!] = \mathtt{fix}_{CPO}\ \mathtt{fun}_{CPO}\ \alpha\ \mathtt{->}\ \mathtt{fun}_{CPO}\ \sigma\ \mathtt{->}\ \mathtt{if}_{CPO}(\mathtt{app}_{CPO}([\![B]\!], \sigma), \mathtt{app}_{CPO}(\alpha, \mathtt{app}_{CPO}([\![S]\!], \sigma)), \sigma)$

// Programs:

$[\![\mathtt{int}\ \mathit{Xl}\mathtt{;}\ S]\!] = \mathtt{app}_{CPO}([\![S]\!], (\mathit{Xl} \mapsto 0))$

Figure 3.21: $\mathcal{R}_{\text{DENOT(IMP)}}$: Denotational semantics of IMP in equational/rewrite logic.

the substitution) on the new mathematical domains without any additional definitional effort, we collapse all the mathematical domains under CPO; in other words, we now have only one large mathematical domain, CPO, which includes the domains of integer numbers, Booleans and states as subdomains.

There is not much to say about the equations in Figure 3.21; they restate the mathematical definitions in Figure 3.20 using our particular CPO language. The equational formalization of the CPO domain in Section 2.6.5 propagates undefinedness through the CPO operations according to their evaluation strategies. This allows us to ignore some cases, such as the last case in the denotation of the assignment or the conditional in Figure 3.20; indeed, if $\mathtt{app}_{CPO}(\llbracket A \rrbracket, \sigma)$ is undefined in some state σ then $\llbracket X = A\,; \rrbracket\sigma$ will also be undefined, because $\mathtt{app}_{CPO}(\ldots, \bot)$ equals $\bot$ according to our CPO formalization in Section 2.6.5.

★ Denotational Semantics of IMP in Maude

Figure 3.22 shows the Maude module corresponding to the rewrite theory in Figure 3.21.

3.4.3 Notes

Denotational semantics is the oldest semantic approach to programming languages. The classic paper that introduced denotational semantics as we know it today, then called "mathematical semantics", was published in 1971 by Scott and Strachey [70]. However, Strachey's interest in the subject started much earlier; he published two papers in 1966 and 1967, [77] and [78], respectively, which mark the beginning of denotational semantics. Before Strachey and Scott published their seminal paper [70], Scott published in 1970 a paper which founded what we call today *domain theory* [71]. Initially, Scott formalized domains as complete lattices (which also admit a fixed-point theorem); in time, bottomed complete partial orders (CPOs, see Section 2.6) turned out to have better properties and they eventually replaced the complete latices.

We have only used very simple domains in our semantics of IMP in this section, such as domains of integers, Booleans, and partial functions. Moreover, for simplicity, we are not going to use complex domains for the IMP++ extension in Section 3.5 either. However, complex languages or better semantics may require more complex domains. In fact, choosing the right domains is one of the most important aspects of a denotational semantics. Poor domains may lead to behavioral limitations or to non-modular denotational semantic definitions. There are two additional important contributions to domain theory that are instrumental in making denotational semantics more usable:

- *Continuation domains.* The use of *continuations* in denotational semantics was proposed in 1974, in a paper by Strachey and Wadsworth [79]. Wadsworth was the one who coined the term "continuation", as representing "the meaning of the rest of the program". Continuations allow to have direct access to the execution flow, in particular to modify it, as needed for the semantics of abrupt termination, exceptions, or call/cc (Scheme was the first language to support call/cc). This way, continuations bring modularity and elegance to denotational definitions. However, they come at a price: using continuations affects the entire language definition (so one needs to change almost everything) and the resulting semantics are harder to read and reason about. There are countless uses of continuations in the literature, not only in denotational semantics; we refer the interested reader to a survey paper by Reynolds, which details continuations and their discovery from various perspectives [63].

- *Powerdomains.* The usual domains of partial functions that we used in our denotational semantics of IMP are not sufficient to define non-deterministic and/or concurrent languages. Consider, for example, the denotation of statements, which are partial functions from states to states. If the language is non-deterministic or concurrent, then a statement may take a state into any of many possible different

```
mod IMP-SEMANTICS-DENOTATIONAL is including IMP-SYNTAX + STATE + CPO .
  sort Syntax .
  subsorts AExp BExp Stmt Pgm < Syntax .
  subsorts Int Bool State < CPO .

  op [[_]] : Syntax  -> CPO .    --- Syntax interpreted in CPOs

  var X : Id . var Xl : List{Id} .  var I : Int .  var A1 A2 A : AExp .
  var T : Bool .  var B1 B2 B : BExp .  var S1 S2 S : Stmt .
  ops sigma alpha arg : -> CPOVar .

  eq [[I]] = funCPO sigma -> I .

  eq [[X]] = funCPO sigma -> _‘(_‘)(sigma,X) .

  eq [[A1 + A2]] = funCPO sigma -> appCPO([[A1]],sigma) +Int appCPO([[A2]],sigma) .

  eq [[A1 / A2]] = funCPO sigma -> ifCPO(appCPO([[A2]],sigma) =/=Bool 0,
                                         appCPO([[A1]],sigma) /Int appCPO([[A2]],sigma),
                                         undefined) .

  eq [[T]] = funCPO sigma -> T .

  eq [[A1 <= A2]] = funCPO sigma -> appCPO([[A1]],sigma) <=Int appCPO([[A2]],sigma) .

  eq [[! B]] = funCPO sigma -> notBool appCPO([[B]],sigma) .

  eq [[B1 && B2]] = funCPO sigma -> ifCPO(appCPO([[B1]],sigma),appCPO([[B2]],sigma),false) .

  eq [[{}]] = funCPO sigma -> sigma .

  eq [[{S}]] = funCPO sigma -> appCPO([[S]],sigma) .

  eq [[X = A ;]]
   = funCPO sigma
     -> appCPO(funCPO arg
               -> ifCPO(_‘(_‘)(sigma,X) =/=Bool undefined, sigma[arg / X], undefined),
               appCPO([[A]],sigma)) .

  eq [[S1 S2]] = funCPO sigma -> appCPO([[S2]],appCPO([[S1]],sigma)) .

  eq [[if (B) S1 else S2]]
   = funCPO sigma -> ifCPO(appCPO([[B]],sigma),appCPO([[S1]],sigma),appCPO([[S2]],sigma)) .

  eq [[while (B) S]]
   = fixCPO(funCPO alpha
            -> funCPO sigma
               -> ifCPO(appCPO([[B]],sigma),appCPO(alpha,appCPO([[S]],sigma)),sigma)) .

  eq [[(int Xl ; S)]] = appCPO([[S]],(Xl |-> 0)) .
endm
```

Figure 3.22: The denotational semantics of IMP in Maude

states, or, said differently, it may take a state into a *set* of states. To give denotational semantics to such languages, Plotkin proposed and formalized the notion of *powerdomain* [59]; the elements of a powerdomain are sets of elements of an underlying domain. Powerdomains make it thus possible to give denotational semantics to non-deterministic languages; used in combination with "resumptions", powerdomains can also be used to give interleaving semantics to concurrent languages.

As seen above, most of the denotational semantics ideas and principles have been proposed and developed in the 1970s. While it is recommended to read the original papers for historical reasons, some of them may actually use notational conventions and constructions which are not in current use today, making them less accessible. The reader interested in a more modern presentation of denotational semantics and domain theory is referred to Schmidt's denotational semantics book [69], and to Mosses' denotational semantics chapter [50] and Gunter and Scott's domain theory chapter [30] in the Handbook of Theoretical Computer Science (1990).

Denotational semantics are commonly defined and executed using functional languages. A particularly appealing aspect of functional languages is that the domains of partial functions, which are crucial for almost any denotational semantics, and fixed-point operators for them can be very easily defined using the already existing functional infrastructure of these languages; in particular, one needs to define no λ-like calculus as we did in Section 2.6.5. There is a plethora of works on implementing and executing denotational semantics on functional languages. We here only mention Papaspyrou's denotational semantics of C [57], implemented in Haskell; it uses about 40 domains in total and spreads over about 5000 lines of Haskell code. An additional advantage of defining denotational semantics in functional languages is that one can relatively easily port them into theorem provers and then prove properties or meta-properties about them. We refer the interested reader to Nipkow [55] for a simple example of how this is done in the context of the Isabelle/HOL prover.

An insightful exercise is to regard domain theory and denotational semantics through the lenses of *initial algebra semantics* [26], which was proposed by Goguen *et al.* in 1977. The initial algebra semantics approach is simple and faithful to equational logic (Section 2.4); it can be summarized with the following steps:

1. Define a language syntax as an algebraic signature, say Σ, which admits an initial (term) algebra T_Σ;
2. Define any needed semantic domain as the carrier of corresponding sort in a multi-sorted set, say D;
3. Give D a Σ-algebra structure, by defining operations corresponding to all symbols in Σ;
4. Conclude that the meaning of the language in D is the unique morphism $D_\_ : T_\Sigma \to D$, which gives meaning D_t in D to any fragment of program t.

Let us apply the initial algebra semantics steps above to IMP:

1. Σ is the signature in Figure 3.2;
2. D_{AExp} is the CPO $(State \rightharpoonup Int, \leq, \bot)$ and similarly for the other sorts;
3. $D_{\_+\_} : D_{AExp} \times D_{AExp} \to D_{AExp}$ is the (total) function defined as $D_{\_+\_}(f_1, f_2)(\sigma) = f_1(\sigma) +_{Int} f_2(\sigma)$ for all $f_1, f_2 \in D_{AExp}$ and $\sigma \in State$, and similarly for the other syntactic constructs.
4. The meaning of IMP is given by the unique morphism $D_\_ : T_\Sigma \to D_{AExp}$.

Therefore, one can regard a denotational semantics of a language as an initial algebra semantics applied to a *particular* algebra D; in our IMP case, for example, what we defined as $[\![a]\!]$ for $a \in AExp$ in denotational semantics is nothing but D_a. Moreover, we can now *prove* our previous denotational semantics definitions; for example, we can prove $D_{a_1+a_2}(\sigma) = D_{a_1}(\sigma) +_{Int} D_{a_2}(\sigma)$. The above was a very brief account of initial

algebra semantics, but sufficient to appreciate the foundational merits of initial algebra semantics in the context of denotational semantics (initial algebra semantics has many other applications). From an initial algebra semantics perspective, a denotational semantics is all about defining an algebra D in which the syntax is interpreted. How each fragment gets a meaning follows automatically, from more basic principles. In that regard, initial algebra semantics achieves a cleaner separation of concerns: syntax is defined as a signature, and semantics is defined as an algebra. There are no equations mixing syntax and semantics, like $[\![a_1 + a_2]\!]\sigma = [\![a_1]\!]\sigma +_{Int} [\![a_2]\!]\sigma$.

We are not aware of any other approaches to define denotational semantics using rewriting and then executing it on rewrite engines as we did in Section 3.4.2. While this is not difficult in principle, as seen in this section, it requires one to give executable rewriting definitions of semantic domains and of fixed points. This may be a tedious and repetitive process on simplistic rewrite engines; for example, the use of membership equational logic, which allows computations to take place also on terms whose intermediate sorts cannot be determined, was crucial for our formalization in Section 2.6.5. Perhaps the closest approach to ours is the one by Goguen and Malcolm in [29]; they define a simple imperative language using the OBJ system (a precursor of Maude) and a style which is a mixture of initial algebra semantics and operational semantics. For example, no fixed-points are used in [29], the loops being simply unrolled like in small-step SOS (see Section 3.3).

3.4.4 Exercises

Prove the following exercises, all referring to the IMP denotational semantics in Figure 3.20.

Exercise 77. *Show the associativity of the addition expression construct, that is, that*

$$[\![(a_1 + a_2) + a_3]\!] = [\![a_1 + (a_2 + a_3)]\!]$$

for any $a_1, a_2, a_3 \in AExp$.

Exercise 78. *Show the associativity of the statement sequential composition, that is, that*

$$[\![s_1\ (s_2\ s_3)]\!] = [\![(s_1\ s_2)\ s_3]\!]$$

for any $s_1, s_2, s_3 \in Stmt$. Compare the elegance of formulating and proving this result using denotational semantics with the similar task using small-step SOS (see Exercise 71).

Exercise 79. *State and prove the (correct) distributivity property of division over addition.*

Exercise 80. *Prove that the sequential composition statement "*`if(b)` s_1 `else` s_2 s*" (i.e., the conditional statement composed sequentially with s) and the conditional statement "*`if(b)`$\{\ s_1\ s\ \}$`else`$\{\ s_2\ s\ \}$*" are equivalent, where s_1 and s_2 are blocks and where s is any statement.*

Exercise 81. *Prove that the functions $\mathcal{F} : (State \rightharpoonup State) \to (State \rightharpoonup State)$ associated to* IMP `while` *loops satisfy the hypotheses of the fixed-point Theorem 12, so that the denotation of* IMP `while` *loops is indeed well-defined. Also, prove that the partial functions $w_k : State \rightharpoonup State$ defined as*

$$w_k(\sigma) = \begin{cases} [\![s]\!]^i\sigma & \text{if there is } 0 \leq i < k \text{ s.t. } [\![b]\!][\![s]\!]^i\sigma = \texttt{false} \text{ and } [\![b]\!][\![s]\!]^j\sigma = \texttt{true} \text{ for all } 0 \leq j < i \\ \bot & \text{otherwise} \end{cases}$$

are well-defined, that is, that if an i as above exists then it is unique. Then prove that $w_k = \mathcal{F}^k(\bot)$.

Exercise 82. *Describe all the fixed-points of the function $\mathcal{F}$ associated to the* IMP `while` *loop.*

Exercise 83. *The semantics in Figure 3.20 evaluates a program performing a division-by-zero to* $\bot$*. Modify the denotational semantics of* IMP *so that it returns a state like for correct programs, namely the state in which the division-by-zero took place. In other words, we want a variant of* IMP *where programs fail silently when they perform illegal operations.*

Bibliography

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[2] Leo Bachmair, Ta Chen, I. V. Ramakrishnan, Siva Anantharaman, and Jacques Chabin. Experiments with associative-commutative discrimination nets. In *IJCAI*, pages 348–355, 1995.

[3] J.-P. Banâtre, A. Coutant, and D. Le Métayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4(2):133 – 144, 1988.

[4] Jean-Pierre Banâtre and Daniel Le Métayer. A new computational model and its discipline of programming. Technical Report INRIA-RR–566, Institut National de Recherche en Informatique et en Automatique (INRIA), 35 - Rennes (France), 1986.

[5] Jean-Pierre Banâtre and Daniel Le Métayer. Chemical reaction as a computational model. In *Functional Programming*, Workshops in Computing, pages 103–117. Springer, 1989.

[6] Jean-Pierre Banâtre and Daniel Le Métayer. The gamma model and its discipline of programming. *Sci. Comput. Program.*, 15(1):55–77, 1990.

[7] Jan Bergstra and J. V. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras. *Journal of the Association for Computing Machinery*, 42(6):1194–1230, 1995.

[8] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *POPL*, pages 81–94, 1990.

[9] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[10] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN *V 3.4 User Manual*. LORIA, Nancy (France), fourth edition, January 2000.

[11] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[12] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.

[13] Christiano Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Departamento de Informática, Pontificia Universidade Católica de Rio de Janeiro, Brasil, 2001.

[14] Christiano Braga and José Meseguer. Modular rewriting semantics in practice. *Electr. Notes Theor. Comput. Sci.*, 117:393–416, 2005.

[15] Christiano de O. Braga, E. Hermann Hæusler, José Meseguer, and Peter D. Mosses. Mapping modular sos to rewriting logic. In *LOPSTR'02: Proceedings of the 12th international conference on Logic based program synthesis and transformation*, pages 262–277, Berlin, Heidelberg, 2003. Springer-Verlag.

[16] Fabricio Chalub and Christiano Braga. Maude MSOS tool. In Grit Denker and Carolyn Talcott, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 133–146. Elsevier, 2007.

[17] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[18] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[19] Oliver Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Technical Report BRICS RS-04-26, University of Aarhus, November 2004.

[20] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In *Second International Workshop on Rule-Based Programming (RULE 2001)*, volume 59(4) of *ENTCS*, pages 358–374, 2001.

[21] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[22] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer, 1979.

[23] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Rosu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

[24] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[25] Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[26] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24:68–95, January 1977.

[27] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985. Preliminary versions have appeared in: *SIGPLAN Notices*, July 1981, Volume 16, Number 7, pages 24–37; SRI Computer Science Lab, Report CSL-135, May

1982; and Report CSLI-84-15, Center for the Study of Language and Information, Stanford University, September 1984.

[28] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

[29] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Foundations of Computing. The MIT Press, May 1996.

[30] Carl A. Gunter and Dana S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 633–674. MIT Press / Elsevier, 1990.

[31] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and validation methods*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[32] Matthew Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, New York, N.Y., 1990.

[33] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[34] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[35] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd. Edition*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Tech. Report SRI-CSL-93-05, August 1993. Second published in Electronic Notes in Theoretical Computer Science, Volume 4, 1996.

[36] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[37] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proceedings of 15th International Conference on Rewriting Techniques and Applications, (RTA'04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 301–311, 2004.

[38] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 301–311. Springer, 2004.

[39] José Meseguer. Conditional rewriting logic: Deduction, models and concurrency. In *Conditional and Typed Rewriting Systems (CTRS'90)*, volume 516 of *Lecture Notes in Computer Science*, pages 64–91. Springer, 1990.

[40] José Meseguer. A logical theory of concurrent objects. In *OOPSLA/ECOOP*, pages 101–115, 1990.

[41] José Meseguer. Rewriting as a unified model of concurrency. In *Theories of Concurrency: Unification and Extension (CONCUR'90)*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer, 1990.

[42] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[43] Josè Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer Berlin / Heidelberg, 1996.

[44] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2004.

[45] José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In *Proceedings of the 2nd International Joint Conference on Automated Reasoning (IJCAR'04)*, volume 3097 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2004.

[46] José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2004.

[47] José Meseguer and Grigore Rosu. The rewriting logic semantics project. *J. TCS*, 373(3):213–237, 2007. Also appeared in *SOS '05*, volume 156(1) of *ENTCS*, pages 27–56, 2006.

[48] José Meseguer and Grigore Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[49] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.

[50] Peter D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 575–631. MIT Press / Elsevier, 1990.

[51] Peter D. Mosses. Foundations of modular sos. In Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *MFCS*, volume 1672 of *Lecture Notes in Computer Science*, pages 70–80. Springer, 1999.

[52] Peter D. Mosses. Pragmatics of modular SOS. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.

[53] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.

[54] Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009.

[55] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics. *Formal Asp. Comput.*, 10(2):171–186, 1998.

[56] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[57] Nikolaos S. Papaspyrou. Denotational semantics of ANSI C. *Computer Standards and Interfaces*, 23(3):169–185, 2001.

[58] Gheorghe Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 2000.

[59] G. D. Plotkin. A powerdomain construction. *SIAM J. of Computing*, 5(3):452–487, September 1976.

[60] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981. Republished in Journal of Logic and Algebraic Programming, Volume 60-61, 2004.

[61] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[62] Emil L. Post. Finite combinatory processes—formulation 1. *The Journal of Symbolic Logic*, 1(3):pp. 103–105, 1936.

[63] John C. Reynolds. The discoveries of continuations. *Lisp Symbolic Computation*, 6:233–248, November 1993.

[64] Grigore Rosu. Cs322, fall 2003 - programming language design: Lecture notes. Technical Report UIUCDCS-R-2003-2897, University of Illinois at Urbana-Champaign, Department of Computer Science, December 2003. Lecture notes of a course taught at UIUC.

[65] Grigore Rosu. Equality of streams is a pi_2^0-complete problem. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*. ACM, 2006.

[66] Grigore Rosu. K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, Computer Science Department, University of Illinois at Urbana-Champaign, 2006. A previous version of this work has been published as technical report UIUCDCS-R-2005-2672 in 2005. K was first introduced in 2003, in the technical report UIUCDCS-R-2003-2897: lecture notes of CS322 (programming language design).

[67] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability*. MIT press, Cambridge, MA, 1987.

[68] Grigore Rosu and Traian Florin Serbănuță. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[69] David A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[70] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford University Computing Laboratory, 1971.

[71] Dana S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG–2, Oxford University Computing Laboratory, Oxford, England, November 1970.

[72] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In Werner Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 247–260. Springer, 1992.

[73] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM J. Comput.*, 24(6):1207–1234, 1995.

[74] Traian Florin Serbănută, Grigore Rosu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207:305–340, 2009.

[75] Traian Florin Serbănută, Gheorghe Stefănescu, and Grigore Rosu. Defining and executing P systems with structured data in K. In David W. Corne, Pierluigi Frisco, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Workshop on Membrane Computing (WMC'08)*, volume 5391 of *Lecture Notes in Computer Science*, pages 374–393. Springer, 2009.

[76] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[77] Christopher Strachey. Towards a formal semantics. In *Proceedings of IFIP TC2 Working Conference on Formal Language Description Languages for Computer Programming*, pages 198–220. North Holland, Amsterdam, 1966.

[78] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 2000. Lecture Notes for a 1967 NATO International Summer School in Computer Programming, Copenhagen; also available from Programming Research Group, University of Oxford, August 1967.

[79] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13:135–152, 2000. Reprinted version of 1974 Programming Research Group Technical Monograph PRG-11, Oxford University Computing Laboratory.

[80] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1937.

[81] Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, 2002.

[82] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.

[83] Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in Maude. Technical Report 134-03, Departamento de Sistemas Informàticos y Programaciòn, Universidad Complutense de Madrid, 2003.

[84] Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in maude. *J. Log. Algebr. Program.*, 67(1-2):226–293, 2006.

[85] Eelco Visser. Program Transf. with Stratego/XT: Rules, Strategies, Tools, and Systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.

[86] Philip Wadler. The essence of functional programming. In *"Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages"*, ACM, pages 1–14, 1992.

[87] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

[88] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[89] Yong Xiao, Zena M. Ariola, and Michael Mauny. From syntactic theories to interpreters: A specification language and its compilation. In *First International Workshop on Rule-Based Programming (RULE 2000)*, 2000.

[90] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating the proof of unique decomposition. *Higher Order and Symbolic Computation*, 14(4):387–409, 2001.