3.6 Modular Structural Operational Semantics (MSOS)

Modular structural operational semantics (MSOS) was introduced, as its name implies, to address the non-modularity aspects of (big-step and/or small-step) SOS. As already seen in Section 3.5, there are several reasons why big-step and small-step SOS are non-modular, as well as several facets of non-modularity in general. In short, a definitional framework is *non-modular* when, in order to add a new feature to an existing language or calculus, one needs to revisit and change some or all of the already defined unrelated features. For example, recall the IMP extension with input/output in Section 3.5.2. We had to add new semantic components in the IMP configurations, both in the big-step and in the small-step SOS definitions, to hold the input/output buffers. That meant, in particular, that *all* the existing big-step and/or small-step SOS rules of IMP had to change. That was, at best, very inconvenient.

Before we get into the technicalities of MSOS, one natural question to address is why we need modularity of language definitions. One may argue that defining a programming language is a major endeavor, done once and for all, so having to go through the semantic rules many times is, after all, not such a bad idea, because it gives one the chance to find and fix potential errors in them. Here are several reasons why modularity is desirable in language definitions, in no particular order:

- Having to modify many or all rules whenever a new rule is added that modifies the structure of the configuration is actually more error prone than it may seem, because rules become heavier to read and debug; for example, one can write σ instead of σ' in a right-hand side of a rule and a different or wrong language is defined.

- A modular semantic framework allows us to more easily reuse semantics of existing and probably already well-tested features in other languages or language extensions, thus increasing our productivity as language designers and our confidence in the correctness of the resulting semantic language definition.

- When designing a new language, as opposed to an existing well-understood language, one needs to experiment with features and combinations of features; having to do lots of unrelated changes whenever a new feature is added to or removed from the language burdens the language designer with boring tasks taking considerable time that could have been otherwise spent on actual interesting language design issues.

- There is a plethora of domain-specific languages these days, generated by the need to abstract away from low-level programming language details to important, domain-specific aspects of the application. Therefore, there is a need for rapid language design and experimentation for various domains. Moreover, domain-specific languages tend to be very dynamic, being added or removed features frequently as the domain knowledge evolves. It would be very nice to have the possibility to "drag-and-drop" language features in one's language, such as functions, exceptions, objects, etc.; however, in order for that to be possible, modularity is crucial.

Whether MSOS gives us such a desired and general framework for modular language design is and will probably always be open for debate. However, to our knowledge, MSOS is the first framework that explicitly recognizes the importance of modular language design and provides explicit support to achieve it in the context of SOS. Reduction semantics with evaluation contexts (see Section 3.7) was actually proposed before MSOS and also offers modularity in language semantic definitions, but its modularity comes as a consequence of a different way to propagate reductions through language constructs and not as an explicit goal that it strives to achieve.

There are both big-step and small-step variants of MSOS, but we discuss only small-step MSOS. We actually generically call MSOS the small-step, implicitly-modular variant of MSOS (see Section 3.6.4). To bring modularity to SOS, MSOS proposes the following:

- Separate the syntax (i.e., the fragment of program under consideration) from the non-syntactic components in configurations, and treat them differently, as explained below;
- Make the transitions relate syntax to syntax only (as opposed to configurations), and hide the non-syntactic components in a transition label, as explained below;
- Encode in the transition label all the changes in the non-syntactic components of the configuration that need to be applied together with the syntactic reduction given by the transition;
- Use specialized notation in transition labels together with a discipline to refer to the various semantic components and to say that some of them stay unchanged; also, labels can be explicitly or implicitly shared by the conditions and the conclusion of a rule, elegantly capturing the idea that "changes are propagated" through desired language constructs.

A transition in MSOS is of the form

$$P \xrightarrow{\Delta} P'$$

where P and P' are programs or fragments of programs and Δ is a *label describing the semantic configuration components both before and after the transition.* Specifically, Δ is a record containing fields denoting the semantic components of the configuration. The preferred notation in MSOS for stating that in label Δ the semantic component associated to the field name field *before* the transition takes place is α is $\Delta = \{\mathsf{field} = \alpha, \ldots\}$. Similarly, the preferred notation for stating that the semantic component associated to field field *after* the transition takes place is β is $\Delta = \{\mathsf{field'} = \beta, \ldots\}$ (the field name is primed). For example, the second MSOS rule for variable assignment (when the assigned arithmetic expression is already evaluated) is (this is rule (MSOS-Asgn) in Figure 3.24):

$$x := i \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state'}=\sigma[i/x],\ \ldots\}} \mathtt{skip} \qquad \text{if } \sigma(x) \neq \bot$$

It is easy to desugar the rule above into a more familiar SOS rule of the form:

$$\langle x := i, \sigma\rangle \to \langle\, \mathtt{skip}\,, \sigma[i/x]\rangle \qquad \text{if } \sigma(x) \neq \bot$$

The above is precisely the rule (SmallStep-Asgn) in the small-step SOS of IMP (see Figure 3.15). The MSOS rule is actually more modular than the SOS one, because of the "...", which says that everything else in the configuration stays unchanged. For example, if we want to extend the language with input/output language constructs as we did in Section 3.5.2, then new semantic components, namely the input and output buffers, need to be added to the configuration. Moreover, as seen in Section 3.5.2, the SOS rule above needs to be changed into a rule of the form

$$\langle x := i, \sigma, \omega_{in}, \omega_{out}\rangle \to \langle \mathtt{skip}, \sigma[i/x], \omega_{in}, \omega_{out}\rangle \qquad \text{if } \sigma(x) \neq \bot$$

where ω_{in} and ω_{out} are the input and output buffers, respectively, which stay unchanged during the variable assignment operation, while the MSOS rule does not need to be touched.

To impose a better discipline on the use of labels, at the same time making the notation even more compact, MSOS splits the fields into three categories: *read-only*, *read-write*, and *write-only*. The field `state` above was read-write, meaning that the transition label can both read its value before the transition takes place and write its value after the transition takes place. Unlike the state, which needs to be both read and written, there are semantic configuration components that only need to be read, as well as ones that only need to be written. In these cases, it is recommended to use read-only or write-only fields.

Read-only fields are only inspected by the rule, but not modified, so they only appear unprimed in labels. For example, the following can be one of the MSOS rules for the `let` binding language construct in a pure functional language where expressions yield no side-effects:

$$\frac{e_2 \xrightarrow{\{\mathsf{env}=\rho[v_1/x],\dots\}} e_2'}{\texttt{let}\,x = v_1\,\texttt{in}\,e_2 \xrightarrow{\{\mathsf{env}=\rho,\dots\}} \texttt{let}\,x = v_1\,\texttt{in}\,e_2'}$$

Indeed, transitions do not modify the environment in a pure functional language. They only use it in a read-only fashion to lookup the variable values. A new environment is created in the premise of the rule above to reduce the body of the `let`, but neither of the transitions in the premise or in the conclusion of the rule change their environment. Note that this was not the case for IMP extended with `let` in Section 3.5.5, because there we wanted blocks and local variable declarations to desugar to `let` statements. Since we wanted blocks to be allowed to modify variables which were not declared locally, like it happens in conventional imperative and object-oriented languages, we needed an impure variant of `let`. As seen in Section 3.6.2, our MSOS definition of IMP++'s `let` uses a read-write attribute (the `state` attribute). We do not discuss read-only fields any further here.

Write-only fields are used to record data that is not analyzable during program execution, such as the output or the trace. Their names are always primed and they have a free monoid semantics—everything written on them is actually added to the end (see [62] for technical details). Consider, for example, the extension of IMP with an output (or print) statement in Section 3.5.2, whose MSOS second rule (after the argument is evaluated, namely rule (MSOS-PRINT) in Section 3.6.2) is:

$$\texttt{print(}\,i\,\texttt{)} \xrightarrow{\{\mathsf{output'}=i,\ \dots\}} \texttt{skip}$$

Compare the rule above with the one below, which uses a read-write attribute instead:

$$\texttt{print(}\,i\,\texttt{)} \xrightarrow{\{\mathsf{output}=\omega_{out},\ \mathsf{output'}=\omega_{out}{:}i,\ \dots\}} \texttt{skip}$$

Indeed, mentioning the ω_{out} like in the second rule above is unnecessary, error-prone (e.g., one may forget to add it to the primed field or may write $i : \omega_{out}$ instead of $\omega_{out} : i$), and non-modular (e.g., one may want to change the monoid construct, say to write $\omega_{out} \cdot i$ instead of $\omega_{out} : i$, etc.).

MSOS achieves modularity in two ways:

1. By making intensive use of the record comprehension notation "...", which, as discussed, indicates that more fields could follow but that they are not of interest. In particular, if the MSOS rule has no premises, like in the rules for the assignment and print statements discussed above, than the "..." says that the remaining contents of the label stays unchanged after the application of the transition; and

2. By reusing the same label or portion of label both in the premise and in the conclusion of an MSOS proof system rule. In particular, if "..." is used in the labels of both the premise and the conclusion of an MSOS rule, then all the occurrences of "..." stand for the same portion of label, that is, the same fields bound to the same semantic components.

For example, the following MSOS rules for first-statement reduction in sequential composition are equivalent and say that all the configuration changes generated by reducing s_1 to s_1' are propagated when reducing $s_1 \,;\, s_2$ to $s_1' \,;\, s_2$:

$$\frac{s_1 \xrightarrow{\Delta} s_1'}{s_1 \,;\, s_2 \xrightarrow{\Delta} s_1' \,;\, s_2}$$

$$\frac{s_1 \xrightarrow{\{\ldots\}} s_1'}{s_1 \,;\, s_2 \xrightarrow{\{\ldots\}} s_1' \,;\, s_2}$$

$$\frac{s_1 \xrightarrow{\{\mathsf{state}=\sigma,\ \ldots\}} s_1'}{s_1 \,;\, s_2 \xrightarrow{\{\mathsf{state}=\sigma,\ \ldots\}} s_1' \,;\, s_2}$$

Indeed, advancing the first statement in a sequential composition of statements one step has the same effect on the configuration as if the statement was advanced the same one step in isolation, without the other statement involved; said differently, the side effects are all properly propagated.

MSOS (the implicitly-modular variant of it, see Section 3.6.4) has been refined to actually allow for dropping such redundant labels like above from rules. In other words, if a label is missing from a transition then the *implicit label* is assumed: if the rule is unconditional then the implicit label is the identity label (in which the primed fields have the same values as the corresponding unprimed ones, etc.), but if the rule is conditional then the premise and the conclusion transitions share the same label, that is, they perform the same changes on the semantic components of the configuration. With this new notational convention, the most elegant and compact way to write the rule above in MSOS is:

$$\frac{s_1 \rightarrow s_1'}{s_1 \,;\, s_2 \rightarrow s_1' \,;\, s_2}$$

This is precisely the rule (MSOS-Seq-Arg1) in Figure 3.24, part of the MSOS semantics of IMP.

One of the important merits of MSOS is that it captures formally many of the tricks that language designers informally use to avoid writing awkward and heavy SOS definitions.

Additional notational shortcuts are welcome in MSOS if properly explained and made locally rigorous, without having to rely on other rules. For example, the author of MSOS finds the rule

$$x \xrightarrow{\{\mathsf{state},\ \ldots\}} \mathsf{state}(x) \quad \text{if } \mathsf{state}(x) \neq \bot$$

to be an acceptable variant of the lookup rule:

$$x \xrightarrow{\{\mathsf{state}=\sigma,\ \ldots\}} \sigma(x) \quad \text{if } \sigma(x) \neq \bot$$

despite the fact that, strictly speaking, $\mathsf{state}(x)$ does not make sense by itself (recall that state is a field name, not the state) and that field names are expected to be paired with their semantic components in labels. Nevertheless, there is only one way to make sense of this rule, namely to replace any use of state by its semantic contents, which therefore does not need to be mentioned.

A major goal when using MSOS to define languages or calculi is to write on the labels as little information as possible and to use the implicit conventions for the missing information. That is because everything written on labels may work against modularity if the language is later on extended or simplified. As an extreme case, if one uses only read/write fields in labels and mentions all the fields together with all their semantic contents on every single label, then MSOS becomes conventional SOS and therefore suffers from the same limitations as SOS with regards to modularity.

Recall the rules in Figure 3.16 for deriving the transitive closure $\to^\star$ of the small-step SOS relation $\to$. In order for two consecutive transitions to compose, the source configuration of the second had to be identical to the target configuration of the first. A similar property must also hold in MSOS, otherwise one may derive inconsistent computations. This process is explained in MSOS by making use of category theory (see [62] for technical details on MSOS; see Section 2.10 for details on category theory), associating MSOS labels with morphisms in a special category and then using the morphism composition mechanism of category theory.

However, category theory is not needed in order to understand how MSOS works in practice. A simple way to explain its label composition is by translating, or desugaring MSOS definitions into SOS, as we implicitly implied when we discussed the MSOS rule for variable assignment above. Indeed, once one knows all the fields in the labels, which happens once a language definition is complete, one can automatically associate a standard small-step SOS definition to the MSOS one by replacing each MSOS rule with an SOS rule over configurations including, besides the syntactic contents, the complete semantic contents extracted from the notational conventions in the label. The resulting SOS configurations will not have fields anymore, but will nevertheless contain all the semantic information encoded by them. For example, in the context of a language containing only a state and an output buffer as semantic components in its configuration (note that IMP++ contained an input buffer as well), the four rules discussed above for variable assignment, output, sequential composition, and lookup desugar, respectively, into the following conventional SOS rules:

$$\langle x := i, \sigma, \omega\rangle \to \langle\, \texttt{skip}\,, \sigma[i/x], \omega\rangle \quad \text{if } \sigma(x) \neq \bot$$

$$\langle\, \texttt{print(}\, i\, \texttt{)}\,, \sigma, \omega\rangle \to \langle\, \texttt{skip}\,, \sigma, \omega : i\rangle$$

$$\frac{\langle s_1, \sigma, \omega\rangle \to \langle s_1', \sigma', \omega'\rangle}{\langle s_1 \;\texttt{;}\; s_2, \sigma, \omega\rangle \to \langle s_1' \;\texttt{;}\; s_2, \sigma', \omega'\rangle}$$

$$\langle x, \sigma, \omega\rangle \to \langle \sigma(x), \sigma, \omega\rangle \quad \text{if } \sigma(x) \neq \bot$$

Recall that for unconditional MSOS rules the meaning of the missing label fields is "stay unchanged", while in the case of conditional rules the meaning of the missing fields is "same changes in conclusion as in the premise". In order for all the changes explicitly or implicitly specified by MSOS rules to apply, one also needs to provide an initial state for all the attributes, or in terms of SOS, an initial configuration. The initial configuration is often left unspecified in MSOS or SOS paper language definitions, but it needs to be explicitly given when one is concerned with executing the semantics. In our SOS definition of IMP in Section 3.3.2 (see Figure 3.15), we created the appropriate initial configuration in which the top-level statement was executed using the proof system itself, more precisely the rule (SMALLSTEP-VAR) created a configuration holding a statement and a state from a configuration holding only the program. That is not possible in MSOS, because MSOS assumes that the structure of the label record does not change dynamically as the rules are applied. Instead, it assumes all the attributes given and fixed. Therefore, one has to explicitly state the initial values corresponding to each attribute in the initial state. However, in practice those initial values are understood and, consequently, we do not bother defining them. For example, if an attribute holds a list or a set, then its initial value is the empty list or set; if it holds a partial function, then its initial value is the partial function undefined everywhere; etc.

This way of regarding MSOS as a convenient front-end to SOS also supports the introduction of further notational conventions in MSOS if desired, like the one discussed above using $\mathsf{state}(x)$ instead of $\sigma(x)$ in the right-hand side of the transition, provided that one explains how such conventions are desugared when going from MSOS to SOS. Finally, the translation of MSOS into SOS also allows MSOS to borrow from SOS the reflexive/transitive closure $\rightarrow^{\star}$ of the one-step relation.

3.6.1 The MSOS of IMP

Figures 3.23 and 3.24 show the MSOS definition of IMP. There is not much to comment on the MSOS rules in these figures, except, perhaps, to note how compact and elegant they are compared to the corresponding SOS definition in Figures 3.14 and 3.15. Except for the three rules (MSOS-LOOKUP), (MSOS-ASGN), and (MSOS-VAR), which make use of labels, they are as compact as they can be in any SOS-like setting for any language including the defined constructs. Also, the above-mentioned three rules only mention those components from the labels that they really need, so they allow for possible extensions of the language, like the IMP++ extension in Section 3.5.

The rule (MSOS-VAR) is somehow different from the other rules that need the information in the label, in that it uses an attribute which has the type read-write but it only writes it without reading it. This is indeed possible in MSOS. The type of an attribute cannot be necessarily inferred from the way it is used in some of the rules, and not all rules must use the same attribute in the same way. One should explicitly clarify the type of each attribute before one gives the actual MSOS rules, and one is not allowed to change the attribute types dynamically, during derivations. Indeed, if the type of the `output` attribute in the MSOS rules for output above (and also in Section 3.6.2) were read-write, then the rules would wrongly imply that the output buffer will only store the last value, the previous ones being lost (this could be a desirable semantics in some cases, but not here).

Since the MSOS proof system in Figures 3.23 and 3.24 translates, following the informal procedure described above, in the SOS proof system in Figures 3.14 and 3.15, basically all the small-step SOS intuitions and discussions for IMP in Section 3.3.2 carry over here almost unchanged. In particular:

Definition 18. *We say that $C \rightarrow C'$ is derivable with the MSOS proof system in Figures 3.23 and 3.24, written* $\mathrm{MSOS(IMP)} \vdash C \rightarrow C'$, *iff* $\mathrm{SMALLSTEP(IMP)} \vdash C \rightarrow C'$ *(using the proof system in Figures 3.14 and 3.15). Similarly,* $\mathrm{MSOS(IMP)} \vdash C \rightarrow^{\star} C'$ *iff* $\mathrm{SMALLSTEP(IMP)} \vdash C \rightarrow^{\star} C'$.

Note, however, that MSOS is more syntactic in nature than SOS, in that each of its reduction rules requires syntactic terms in both sides of the transition relation. In particular, that means that, unlike in SOS (see Exercise 52), in MSOS one does not have the option to dissolve statements from configurations anymore. Instead, one needs to reduce them to `skip` or some similar syntactic constant; if the original language did not have such a constant then one needs to invent one and add it to the original language or calculus syntax.

$$x \xrightarrow{\{\mathsf{state}=\sigma,\ \ldots\}} \sigma(x) \quad \text{if } \sigma(x) \neq \bot \qquad \text{(MSOS-Lookup)}$$

$$\frac{a_1 \to a_1'}{a_1 \texttt{+} a_2 \to a_1' \texttt{+} a_2} \qquad \text{(MSOS-Add-Arg1)}$$

$$\frac{a_2 \to a_2'}{a_1 \texttt{+} a_2 \to a_1 \texttt{+} a_2'} \qquad \text{(MSOS-Add-Arg2)}$$

$$i_1 \texttt{+} i_2 \to i_1 +_{Int} i_2 \qquad \text{(MSOS-Add)}$$

$$\frac{a_1 \to a_1'}{a_1 \,\texttt{/}\, a_2 \to a_1' \,\texttt{/}\, a_2} \qquad \text{(MSOS-Div-Arg1)}$$

$$\frac{a_2 \to a_2'}{a_1 \,\texttt{/}\, a_2 \to a_1 \,\texttt{/}\, a_2'} \qquad \text{(MSOS-Div-Arg2)}$$

$$i_1 \,\texttt{/}\, i_2 \to i_1 \,/_{Int}\, i_2 \quad \text{if } i_2 \neq 0 \qquad \text{(MSOS-Div)}$$

$$\frac{a_1 \to a_1'}{a_1 \texttt{<=} a_2 \to a_1' \texttt{<=} a_2} \qquad \text{(MSOS-Leq-Arg1)}$$

$$\frac{a_2 \to a_2'}{i_1 \texttt{<=} a_2 \to i_1 \texttt{<=} a_2'} \qquad \text{(MSOS-Leq-Arg2)}$$

$$i_1 \texttt{<=} i_2 \to i_1 \leq_{Int} i_2 \qquad \text{(MSOS-Leq)}$$

$$\frac{b \to b'}{\texttt{not}\, b \to \texttt{not}\, b'} \qquad \text{(MSOS-Not-Arg)}$$

$$\texttt{not true} \to \texttt{false} \qquad \text{(MSOS-Not-True)}$$

$$\texttt{not false} \to \texttt{true} \qquad \text{(MSOS-Not-False)}$$

$$\frac{b_1 \to b_1'}{b_1 \,\texttt{and}\, b_2 \to b_1' \,\texttt{and}\, b_2} \qquad \text{(MSOS-And-Arg1)}$$

$$\texttt{false and}\, b_2 \to \texttt{false} \qquad \text{(MSOS-And-False)}$$

$$\texttt{true and}\, b_2 \to b_2 \qquad \text{(MSOS-And-True)}$$

Figure 3.23: MSOS(IMP) — MSOS of IMP Expressions ($i_1, i_2 \in Int$; $x \in Id$; $a_1, a_1', a_2, a_2' \in AExp$; $b, b', b_1, b_1', b_2 \in BExp$; $\sigma \in State$).

$$\frac{a \to a'}{x := a \to x := a'} \qquad \text{(MSOS-Asgn-Arg2)}$$

$$x := i \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state'}=\sigma[i/x],\ \ldots\}} \mathtt{skip} \quad \text{if } \sigma(x) \neq \bot \qquad \text{(MSOS-Asgn)}$$

$$\frac{s_1 \to s_1'}{s_1 \mathbin{;} s_2 \to s_1' \mathbin{;} s_2} \qquad \text{(MSOS-Seq-Arg1)}$$

$$\mathtt{skip} \mathbin{;} s_2 \to s_2 \qquad \text{(MSOS-Seq-Skip)}$$

$$\frac{b \to b'}{\mathtt{if}\, b\, \mathtt{then}\, s_1\, \mathtt{else}\, s_2 \to \mathtt{if}\, b'\, \mathtt{then}\, s_1\, \mathtt{else}\, s_2} \qquad \text{(MSOS-If-Arg1)}$$

$$\mathtt{if\ true\ then}\, s_1\, \mathtt{else}\, s_2 \to s_1 \qquad \text{(MSOS-If-True)}$$

$$\mathtt{if\ false\ then}\, s_1\, \mathtt{else}\, s_2 \to s_2 \qquad \text{(MSOS-If-False)}$$

$$\mathtt{while}\, b\, \mathtt{do}\, s \to \mathtt{if}\, b\, \mathtt{then}\, (s \mathbin{;} \mathtt{while}\, b\, \mathtt{do}\, s)\, \mathtt{else\ skip} \qquad \text{(MSOS-While)}$$

$$\mathtt{var}\, xl \mathbin{;} s \xrightarrow{\{\mathsf{state'}=\ xl \mapsto 0,\ \ldots\}} s \qquad \text{(MSOS-Var)}$$

Figure 3.24: MSOS(IMP) — MSOS of IMP Statements ($i \in Int$; $x \in Id$; $xl \in \mathbf{List\{}Id\mathbf{\}}$;$a, a' \in AExp$; $b, b' \in BExp$; $s, s_1, s_1', s_2 \in Stmt$; $\sigma \in State$).

3.6.2 The MSOS of IMP++

We next discuss the MSOS of IMP++, playing the same language design scenario as in Section 3.5: we first add each feature separately to IMP, as if that feature was the final extension of the language, and then we add all the features together and investigate the modularity of the resulting definition as well as possibly unexpected feature interactions.

Variable Increment

In MSOS, one can define the increment modularly:

$$\mathtt{++}\, x \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state'}=\sigma[(\sigma(x)+_{Int}1)/x],\ \ldots\}} \sigma(x) +_{Int} 1 \qquad \text{(MSOS-Inc)}$$

No other rule needs to be changed, because MSOS already assumes that, unless otherwise specified, each rule propagates all the configuration changes in its premise(s).

Input/Output

MSOS can modularly support the input/output extension of IMP. We need to add new label attributes holding the input and the output buffers, say input and output, respectively, and then to add the corresponding rules for the input/output constructs. Note that the input attribute is read-write, while the output attribute is write-only. Here are the MSOS rules for input/output:

$$\mathtt{read()} \xrightarrow{\{\mathsf{input}=i{:}\omega,\ \mathsf{input'}=\omega,\ \ldots\}} i \qquad \text{(MSOS-Read)}$$

$$\frac{a \to a'}{\texttt{print(}\,a\,\texttt{)} \to \texttt{print(}\,a'\,\texttt{)}} \qquad \text{(MSOS-Print-Arg)}$$

$$\texttt{print(}\,i\,\texttt{)} \xrightarrow{\{\mathsf{output'}=i,\ \ldots\}} \texttt{skip} \qquad \text{(MSOS-Print)}$$

Note that, since output is a write-only attribute, we only need to mention the new value that is added to the output in the label of the second rule above. If output was declared as a read-write attribute, then the label of the second rule above would have been $\{\mathsf{output} = \omega, \mathsf{output'} = \omega : i, \ldots\}$. A major implicit objective of MSOS is to minimize the amount of information that the user needs to write in each rule. Indeed, anything written by a user can lead to non-modularity and thus work against the user when changes are performed to the language. For example, if for some reason one declared output as a read-write attribute and then later on one decided to change the list construct for the output integer list from colon "$\_ : \_$" to something else, say "$\_ \cdot \_$", then one would need to change the label in the second rule above from $\{\mathsf{output} = \omega, \mathsf{output'} = \omega : i, \ldots\}$ to $\{\mathsf{output} = \omega, \mathsf{output'} = (\omega \cdot i), \ldots\}$. Therefore, in the spirit of enhanced modularity and clarity, the language designer using MSOS is strongly encouraged to use write-only (or read-only) attributes instead of read-write attributes whenever possible.

Notice the lack of expected duality between the rules (MSOS-Read) and (MSOS-Print) for input and for output above. Indeed, for all the reasons mentioned above, one would like to write the rule (MSOS-Read) more compactly and modularly as follows:

$$\texttt{read()} \xrightarrow{\{\mathsf{input}=i,\ \ldots\}} i$$

Unfortunately, this is not possible with the current set of label attributes provided by MSOS. However, there is no reason why MSOS could not be extended to include more attributes. For example, an attribute called "consumable" which would behave as the dual of write-only, i.e., it would only have an unprimed variant in the label holding a monoid (or maybe a group?) structure like the read-only attributes but it would consume from it whatever is matched by the rule label, would certainly be very useful in our case here. If such an attribute type were available, then our input attribute would be of that type and our MSOS rule for `read()` would be like the one above.

A technical question regarding the execution of the resulting MSOS definition is how to provide input to programs. Or, put differently, how to initialize configurations. One possibility is to assume that the user is fully responsible for providing the initial attribute values. This is, however, rather inconvenient, because the user would then always have to provide an empty state and an empty output buffer in addition to the desired input buffer in each configuration. A more convenient approach is to invent a special syntax allowing the user to provide precisely a program and an input to it, and then to automatically initialize all the attributes with their expected values. Let us pair a program p and an input ω for it using a configuration-like notation of the form $\langle p, \omega \rangle$. Then we can replace the rule (MSOS-Var) in Figure 3.24 with the following rule:

$$\langle \texttt{var}\, xl\ \texttt{;}\ s,\ \omega \rangle \xrightarrow{\{\mathsf{state'}=\, xl \mapsto 0,\ \mathsf{input'}=\omega,\ \ldots\}} s$$

Abrupt Termination

MSOS allows for a more modular semantics of abrupt termination than the more conventional semantic approaches discussed in Section 3.5.3. However, in order to achieve modularity, we need to extend the syntax of IMP with a `top` construct, similarly to the small-step SOS variant discussed in Section 3.5.3. The key to modularity here is to use the labeling mechanism of MSOS to carry the information that a configuration is in a halting status. Let us assume an additional write-only field in the MSOS labels, called halting, which is *true* whenever the program needs to halt, otherwise it is *false*[4]. Then we can add the following two MSOS rules that set the halting field to *true*:

$$i_1 \;\texttt{/}\; 0 \xrightarrow{\{\mathsf{halting}'=true,\ \ldots\}} i_1 \;\texttt{/}\; 0 \qquad \text{(MSOS-Div-By-Zero)}$$

$$\texttt{halt} \xrightarrow{\{\mathsf{halting}'=true,\ \ldots\}} \texttt{halt} \qquad \text{(MSOS-Halt)}$$

As desired, it is indeed the case now that a fact of the form $s \xrightarrow{\{\mathsf{halting}'=true,\ \ldots\}} s'$ is derivable if and only if $s = s'$ and the next executable step in s is either a `halt` statement or a division-by-zero expression. If one does not like keeping the syntax unchanged when an abrupt termination takes place, then one can add a new syntactic construct, say `stuck` like in [62], and replace the right-hand-side configurations above with `stuck`; that does not conceptually change anything in what follows. The setting seems therefore perfect for adding a rule of the form

$$\frac{s \xrightarrow{\{\mathsf{halting}'=true,\ \ldots\}} s}{s \xrightarrow{\{\mathsf{halting}'=false,\ \ldots\}} \texttt{skip}}$$

and declare ourselves done, because now an abruptly terminated statement terminates just like any other statement, with a `skip` statement as result and with a label containing a non-halting status. Unfortunately, that does not work, because such a rule would interfere with other rules taking statement reductions as preconditions, for example with the first precondition of the (MSOS-Seq) rule, and thus hide the actual halting status of the precondition. To properly capture the halting status, we define a top level statement construct like we discussed in the context of big-step and small-step SOS above, say `top` *Stmt*, modify the rule (MSOS-Var) from

$$\texttt{var}\, xl \,\texttt{;}\, s \xrightarrow{\{\mathsf{state}'=\, xl\mapsto 0,\ \ldots\}} s$$

to

$$\texttt{var}\, xl \,\texttt{;}\, s \xrightarrow{\{\mathsf{state}'=\, xl\mapsto 0,\ \mathsf{halting}=false,\ \ldots\}} \texttt{top}\, s$$

to mark the top level statement, and then finally include the following three rules:

$$\frac{s \xrightarrow{\{\mathsf{halting}'=false,\ \ldots\}} s'}{\texttt{top}\, s \xrightarrow{\{\mathsf{halting}'=false,\ \ldots\}} \texttt{top}\, s'} \qquad \text{(MSOS-Top-Normal)}$$

$$\texttt{top skip} \rightarrow \texttt{skip} \qquad \text{(MSOS-Top-Skip)}$$

$$\frac{s \xrightarrow{\{halting'=true,\ \ldots\}} s}{\texttt{top}\, s \xrightarrow{\{\mathsf{halting}'=false,\ \ldots\}} \texttt{skip}} \qquad \text{(MSOS-Top-Halting)}$$

[4]Strictly speaking, MSOS requires that the write-only attributes take values from a free monoid; if one wants to be faithful to that MSOS requirement, then one can replace *true* with some letter word and *false* with the empty word.

The use of a `top` construct like above seems unavoidable if we want to achieve modularity. Indeed, we managed to avoid it in the small-step SOS definition of abrupt termination in Section 3.5.3 (paying one relatively acceptable additional small-step to dissolve the halting configuration), because the halting configurations were explicitly, and thus non-modularly propagated through each of the language constructs, so the entire program reduced to a halting configuration whenever a division by zero or a `halt` statement was encountered. Unfortunately, that same approach does not work with MSOS (unless we want to break its modularity, like in SOS), because the syntax is not mutilated when an abrupt termination occurs. The halting signal is captured by the label of the transition. However, the label does not tell us when we are at the top level in order to dissolve the halting status. Adding a new label to hold the depth of the derivation, or at least whether we are the top or not, would require one to (non-modularly) change it in each rule. The use of an additional `top` construct like we did above appears to be the best trade-off between modularity and elegance here.

Note that, even though MSOS can be mechanically translated into SOS by associating to each MSOS attribute an SOS configuration component, the solution above to support abrupt termination modularly in MSOS is *not* modular when applied in SOS via the translation. That is because adding a new attribute in the label means adding a new configuration component, which already breaks the modularity of SOS. In other words, the MSOS technique above cannot be manually used in SOS to obtain a modular definition of abrupt termination in SOS.

Dynamic Threads

The small-step SOS rule for spawning threads discussed in Section 3.5.4 can be straightforwardly turned into MSOS rules:

$$\frac{s \to s'}{\texttt{spawn}\, s \to \texttt{spawn}\, s'} \qquad \text{(MSOS-Spawn-Arg)}$$

$$\texttt{spawn skip} \to \texttt{skip} \qquad \text{(MSOS-Spawn-Skip)}$$

$$\frac{s_2 \to s_2'}{\texttt{spawn}\, s_1 \;\texttt{;}\; s_2 \to \texttt{spawn}\, s_1 \;\texttt{;}\; s_2'} \qquad \text{(MSOS-Spawn-Wait)}$$

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

Even though the MSOS rules above are conceptually identical to the original small-step SOS rules, they are more modular because, unlike the former, they carry over unchanged when the configuration needs to change. Note that the structural identity stating the associativity of sequential composition, called (MSOS-Seq-Assoc) above, is still necessary.

Local Variables

Section 3.5.5 showed how blocks with local variables can be desugared into a uniform `let` construct, and also gave the small-step SOS rules defining the semantics of `let`. Those rules can be immediately adapted into the following MSOS rules:

$$\frac{a \to a'}{\texttt{let}\; x = a \;\texttt{in}\, s \to \texttt{let}\; x = a' \;\texttt{in}\, s} \qquad \text{(MSOS-Let-Exp)}$$

$$\frac{s \xrightarrow{\{\mathsf{state}=\sigma[i/x],\ \mathsf{state}'=\sigma',\ \ldots\}} s'}{\mathtt{let}\ x = i\ \mathtt{in}\, s \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state}'=\sigma'[\sigma(x)/x],\ \ldots\}} \mathtt{let}\ x = \sigma'(x)\ \mathtt{in}\, s'} \qquad \text{(MSOS-Let-Stmt)}$$

$$\mathtt{let}\ x = i\ \mathtt{in}\ \mathtt{skip} \rightarrow \mathtt{skip} \qquad \text{(MSOS-Let-Done)}$$

Like for the other features, the MSOS rules are more modular than their small-step SOS variants.

Since programs are now just ordinary (closed) expressions and they are now executed in the empty state, the rule (MSOS-Var), namely

$$\mathtt{var}\, xl\ \mathtt{;}\ s \xrightarrow{\{\mathsf{state'}= xl \mapsto 0,\ \ldots\}} s$$

needs to change into a rule of the from

$$s \xrightarrow{\{\mathsf{state'}= \cdot,\ \ldots\}} ?$$

Unfortunately, regardless of what we place instead of "?", such a rule will not work. That is because there is nothing to prevent it to apply to any statement at any step during the reduction. To enforce it to happen only at the top of the program and only at the beginning of the reduction, we can wrap the original program (which is a statement) into a one-element configuration-like term $\langle s \rangle$. Then the rule (MSOS-Var) can be replaced with the following rule:

$$\langle s \rangle \xrightarrow{\{\mathsf{state'}= \cdot,\ \ldots\}} s$$

Putting Them All Together

The modularity of MSOS makes it quite easy to put all the features discussed above together and thus define the MSOS of IMP++. Effectively, we have to do the following:

1. We add the three label attributes used for the semantics of the individual features above, namely the read-write input attribute and the two write-only attributes output and halting.
2. We add all the MSOS rules of all the features above *unchanged* (nice!), except for the rule (MSOS-Var) for programs (which changed several times, anyway).
3. To initialize the label attributes, we add a pairing construct $\langle s, \omega \rangle$ like we did when we added the input/output extension of IMP, where s is a statement (programs are ordinary statements now) and ω is a buffer, and replace the rule (MSOS-Var) in Figure 3.24 with the following:

$$\langle s, \omega \rangle \xrightarrow{\{\mathsf{state'}= xl \mapsto 0,\ \mathsf{input}'=\omega,\ \mathsf{halting}'=\mathit{false},\ \ldots\}} \mathtt{top}\, s$$

It is important to note that the MSOS rules of the individual IMP extensions can be very elegantly combined into one language (IMP++). The rule (MSOS-Var) had to globally change in order to properly initialize the attribute values, but nothing had to be done in the MSOS rules of any of the features in order to put it together with the other MSOS rules of the other features.

Unfortunately, even though each individual feature has its intended semantics, the resulting IMP++ language does not. We still have the same semantic problems with regards to concurrency that we had in the context of small-step SOS in Section 3.5.6. More precisely, the concurrency of `spawn` statements is limited to the blocks in which they appear. For example, a statement of the form $(\texttt{let}\,x=i\,\texttt{in spawn}\,s_1)\,\texttt{;}\,s_2$ does not allow s_2 to be evaluated concurrently with s_1. The statement s_1 has to evaluate completely and then the `let` statement dissolved, before any step in s_2 can be performed. To fix this problem, we would have to adopt one of the solutions proposed in Section 3.5.6 in the context of small-step SOS. Thanks to its modularity, MSOS would make any of those solutions easier to implement than small-step SOS. Particularly, one can use the label mechanism to pass a spawned thread and its execution environment all the way to the top modularly, without having to propagate it explicitly through language constructs.

3.6.3 MSOS in Rewriting Logic

Like big-step and small-step SOS, we can also associate a conditional rewrite rule to each MSOS rule and hereby obtain a rewriting logic theory that faithfully (i.e., step-for-step) captures the MSOS definition. There could be different ways to do this. One way to do it is to first desugar the MSOS definition into a step-for-step equivalent small-step SOS definition as discussed above, and then use the faithful embedding of small-step SOS into rewriting logic discussed in Section 3.3.3. The problem with this approach is that the resulting small-step SOS definition, and implicitly the resulting rewriting logic definition, lack the modularity of the original MSOS definition. In other words, if one wanted to extend the MSOS definition with rules that would require global changes to its corresponding SOS definition (e.g., ones adding new semantic components into the label/configuration), then one would also need to manually incorporate all those global changes in the resulting rewriting logic definition.

We first show that any MSOS proof system, say MSOS, can be mechanically translated into a rewriting logic theory, say $\mathcal{R}_{\mathrm{MSOS}}$, in such a way that two important aspects of the original MSOS definition are preserved: 1) the corresponding derivation relations are step-for-step equivalent, that is, $\mathrm{MSOS} \vdash C \to C'$ if and only if $\mathcal{R}_{\mathrm{MSOS}} \vdash \mathcal{R}_{C\to C'}$, where $\mathcal{R}_{C\to C'}$ is the corresponding syntactic translation of the MSOS sequent $C \to C'$ into a rewriting logic sequent; and 2) $\mathcal{R}_{\mathrm{MSOS}}$ is as modular as MSOS. Second, we apply our generic translation technique to the MSOS formal system MSOS(IMP) defined in Section 3.6.1 and obtain a rewriting logic semantics of IMP that is step-for-step equivalent to and as modular as MSOS(IMP). The modularity of MSOS(IMP) and of $\mathcal{R}_{\mathrm{MSOS(IMP)}}$ will pay off when we extend IMP in Section 3.5. Finally, we show how $\mathcal{R}_{\mathrm{MSOS(IMP)}}$ can be seamlessly defined in Maude, yielding another interpreter for IMP (in addition to those corresponding to the big-step and small-step SOS definitions of IMP in Sections 3.2.3 and 3.3.3).

Computationally and Modularly Faithful Embedding of MSOS into Rewriting Logic

Our embedding of MSOS into rewriting logic is very similar to that of small-step SOS, with one important exception: the non-syntactic components of the configuration are all grouped into a *record*, which is a multiset of *attributes*, each attribute being a pair associating appropriate semantic information to a *field*. This allows us to use multiset *matching* (or matching modulo associativity, commutativity, and identity) in the corresponding rewrite rules to extract the needed semantic information from the record, thus achieving not only a computationally equivalent embedding of MSOS into rewriting logic, but also one with the same degree of modularity as MSOS.

Formally, let us assume an arbitrary MSOS formal proof system. Let *Attribute* be a fresh sort and let *Record* be the sort **Bag{***Attribute***}** (that means that we assume all the infrastructure needed to define records as comma-separated bags, or multisets, of attributes). For each field

Field holding semantic contents *Contents* that appears unprimed or primed in any of the labels on any of the transitions in any of the rules of the MSOS proof system, let us assume an operation "$\mathit{Field} = \_ : \mathit{Contents} \to \mathit{Attribute}$" (the name of this postfix unary operation is "$\mathit{Field} = \_$"). Finally, for each syntactic category *Syntax* used in any transition that appears anywhere in the MSOS proof system, let us define a configuration construct "$\langle \_, \_ \rangle : \mathit{Syntax} \times \mathit{Record} \to \mathit{Configuration}$". We are now ready to define our transformation of an MSOS rule into a rewriting logic rule:

1. Translate it into an SOS rule, as discussed right above Section 3.6.1; we could also go directly from MSOS rules to rewriting logic rules, but we would have to repeat most of the steps from MSOS to SOS that were already discussed;

2. Group all the semantic components in the resulting SOS configurations into a corresponding record, where each semantic component translates into a corresponding attribute using a corresponding label;

3. If the same attributes appear multiple times in a rule and their semantic components are not used anywhere in the MSOS rule, then replace them by a generic variable of sort *Record*;

4. Finally, use the technique in Section 3.3.3 to transform the resulting SOS-like rules into rewrite rules, tagging the left-hand-side configurations with the $\circ$ symbol.

Applying the steps above, the four MSOS rules discussed right above Section 3.6.1 (namely the ones for variable assignment, output, sequential composition of statements, and variable lookup) translate into the following rewriting logic rules:

$$\begin{array}{rl}
\circ\langle X := I, (\mathsf{state} = \sigma, \rho)\rangle \to \langle\, \mathtt{skip}\,, (\mathsf{state} = \sigma[I/X], \rho)\rangle & \\
\circ\langle\, \mathsf{output}\,(i), (\mathsf{output} = \omega,\ \rho)\rangle \to \langle\, \mathtt{skip}\,, (\mathsf{output} = \omega i,\ \rho)\rangle & \\
\circ\langle S_1 \,;\, S_2, \rho\rangle \to \langle S_1' \,;\, S_2, \rho'\rangle & \textbf{if} \quad \circ\langle S_1, \rho\rangle \to \langle S_1', \rho'\rangle \\
\circ\langle X, (\mathsf{state} = \sigma, \rho)\rangle \to \langle \sigma(X), (\mathsf{state} = \sigma, \rho)\rangle &
\end{array}$$

We use the same mechanism as for small-step SOS to obtain the reflexive and transitive many-step closure of the MSOS one-step transition relation. This mechanism was discussed in detail in Section 3.3.3; it essentially consists of adding a configuration marker $\star$ together with a rule "$\star\, \mathit{Cfg} \to \star\, \mathit{Cfg}'$ **if** $\circ\, \mathit{Cfg} \to \mathit{Cfg}'$" iteratively applying the one-step relation.

Theorem 5. ***(Faithful embedding of MSOS into rewriting logic)*** *For any MSOS definition* MSOS*, and any appropriate configurations C and C', the following equivalences hold:*

$$\mathrm{MSOS} \vdash C \to C' \iff \mathcal{R}_{\mathrm{MSOS}} \vdash \circ\overline{C} \to^{1} \overline{C'} \iff \mathcal{R}_{\mathrm{MSOS}} \vdash \circ\overline{C} \to \overline{C'}$$

$$\mathrm{MSOS} \vdash C \to^{\star} C' \iff \mathcal{R}_{\mathrm{MSOS}} \vdash \star\overline{C} \to \star\overline{C'}$$

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

Like for the previous embeddings of big-step and small-step SOS into rewriting logic, let us elaborate on the apparent differences between MSOS and $\mathcal{R}_{\mathrm{MSOS}}$ from a user perspective. The most visible difference is the SOS-like style of writing the rules, namely using configurations instead

sorts:

$\mathit{Attribute}$, Record, $\mathit{Configuration}$, $\mathit{ExtendedConfiguration}$

subsorts and aliases:

$\mathit{Record} = \mathbf{Bag}\{\mathit{Attribute}\}$

$\mathit{Configuration} < \mathit{ExtendedConfiguration}$

operations:

$$\begin{array}{rcll}
\mathsf{state} = \_ & : & \mathit{State} \to \mathit{Attribute} & \text{// more fields can be added by need} \\
\langle \_, \_ \rangle & : & \mathit{AExp} \times \mathit{Record} \to \mathit{Configuration} & \\
\langle \_, \_ \rangle & : & \mathit{BExp} \times \mathit{Record} \to \mathit{Configuration} & \\
\langle \_, \_ \rangle & : & \mathit{Stmt} \times \mathit{Record} \to \mathit{Configuration} & \\
\langle \_ \rangle & : & \mathit{Pgm} \to \mathit{Configuration} & \\
\circ\_ & : & \mathit{Configuration} \to \mathit{ExtendedConfiguration} & \text{// reduce one step} \\
\star\_ & : & \mathit{Configuration} \to \mathit{ExtendedConfiguration} & \text{// reduce all steps}
\end{array}$$

rule:

$\star\, \mathit{Cfg} \to \star\, \mathit{Cfg}'$ **if** $\circ\, \mathit{Cfg} \to \mathit{Cfg}'$ // where $\mathit{Cfg}, \mathit{Cfg}'$ are variables of sort $\mathit{Configuration}$

Figure 3.25: Configurations and infrastructure for the rewriting logic embedding of MSOS(IMP).

of labels, which also led to the inheritance of the $\circ$ mechanism from the embedding of SOS into rewriting logic. Therefore, the equivalent rewriting logic definition is slightly more verbose than the original MSOS definition. On the other hand, it has the advantage that it is more direct than the MSOS definition, in that it eliminates all the notational conventions. Indeed, if we strip MSOS out of its notational conventions and go straight to its essence, we find that that essence is precisely its use of multiset matching to modularly access the semantic components of the configuration. MSOS chose to do this on the labels, using specialized conventions for read-only/write-only/read-write components, while our rewriting logic embedding of MSOS does it in the configurations, uniformly for all semantic components. Where precisely this matching takes place is, in our view, less relevant. What is relevant and brings MSOS its modularity is that multiset matching *does* happen. Therefore, similarly to the big-step and small-step SOS representations in rewriting logic, we conclude that the rewrite theory $\mathcal{R}_{\mathrm{MSOS}}$ *is* MSOS, and *not* an encoding of it.

Like for the previous embeddings of big-step and small-step SOS into rewriting logic, unfortunately, $\mathcal{R}_{\mathrm{MSOS}}$ (and implicitly MSOS) still lacks the main strengths of rewriting logic, namely context-insensitive and parallel application of rewrite rules. Indeed, the rules of $\mathcal{R}_{\mathrm{MSOS}}$ can only apply at the top, sequentially, so these rewrite theories corresponding to the faithful embedding of MSOS follow a rather poor rewriting logic style. Like for the previous embeddings, this is not surprising though and does not question the quality of our embeddings. All it says is that MSOS was not meant to have the capabilities of rewriting logic with regards to context-insensitivity and parallelism; indeed, all MSOS attempts to achieve is to address the lack of modularity of SOS and we believe that it succeeded in doing so. Unfortunately, SOS has several other major problems, which are discussed in Section 3.5.

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

Figure 3.26: $\mathcal{R}_{\text{MSOS(IMP)}}$: the complete MSOS of IMP in rewriting logic.

MSOS of IMP in Rewriting Logic

We here discuss the complete MSOS definition of IMP in rewriting logic, obtained by applying the faithful embedding technique discussed above to the MSOS definition of IMP in Section 3.6.1. Figure 3.25 gives an algebraic definition of configurations as well as needed additional record infrastructure; all the sorts, operations, and rules in Figure 3.25 were already discussed either above or in Section 3.3.3. Figure 3.26 gives the rules of the rewriting logic theory $\mathcal{R}_{\text{MSOS(IMP)}}$ that is obtained by applying the procedure above to the MSOS of IMP in Figures 3.23 and 3.24. Like before, we used the rewriting logic convention that variables start with upper-case letters; if they are greek letters, then we use a similar but larger symbol (e.g., σ instead of σ for variables of sort *State*, or ρ instead of ρ for variables of sort *Record*). The following corollary of Theorem 5 establishes the faithfulness of the representation of the MSOS of IMP in rewriting logic:

Corollary 4. MSOS(IMP) $\vdash C \to C' \iff \mathcal{R}_{\text{MSOS(IMP)}} \vdash \circ\overline{C} \to \overline{C'}$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system MSOS(IMP) and generic rewriting logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{MSOS(IMP)}}$; the two are faithfully equivalent. Moreover, by the discussion following Theorem 5, $\mathcal{R}_{\text{MSOS(IMP)}}$ is also as modular as MSOS(IMP). This will be further emphasized in Section 3.5, where we will extend IMP with several features, some of which requiring more attributes.

☆ Maude Definition of IMP MSOS

Figure 3.27 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{MSOS(IMP)}}$ in Figures 3.26 and 3.25. The Maude module `IMP-SEMANTICS-MSOS` in Figure 3.27 is executable, so Maude, through its rewriting capabilities, yields an MSOS interpreter for IMP the same way it yielded big-step and small-step SOS interpreters in Sections 3.2.3 and 3.3.3, respectively; for example, the Maude rewrite 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: 7132 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < skip, state = (n |-> 0 , s |-> 5050) >
```

Note that the rewrite command above took the same number of rewrite steps as the similar command executed on the small-step SOS of IMP in Maude discussed in Section 3.3.3, namely 7132. This is not unexpected, because matching is not counted as rewrite steps, no matter how complex it is.

Like for the big-step and small-step SOS definitions in Maude, one can also use any of the general-purpose tools provided by Maude on the MSOS 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. The same number of states as in the case of small-step SOS will be generated by this search command, namely 1509.

```
mod IMP-CONFIGURATIONS-MSOS is including IMP-SYNTAX + STATE .
  sorts Attribute Record Configuration ExtendedConfiguration .
  subsort Attribute < Record .
  subsort Configuration < ExtendedConfiguration .
  op empty : -> Record .
  op _,_ : Record Record -> Record [assoc comm id: empty] .
  op state`=_ : State -> Attribute .
  op <_,_> : AExp Record -> Configuration .
  op <_,_> : BExp Record -> Configuration .
  op <_,_> : Stmt Record -> 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-MSOS is including IMP-CONFIGURATIONS-MSOS .
  var X : Id .   var R R' : Record . var Sigma Sigma' : State .
  var I I1 I2 : Int .  var Xl : List{Id} .  var S S1 S1' S2 : Stmt .
  var A A' A1 A1' A2 A2' : AExp .  var B B' B1 B1' B2 B2' : BExp .

 crl o < X,(state = Sigma, R) > => < Sigma(X),(state = Sigma, R) >
  if Sigma(X) =/=Bool undefined .
 crl o < A1 + A2,R > => < A1' + A2,R' > if o < A1,R > => < A1',R' > .
 crl o < A1 + A2,R > => < A1 + A2',R' > if o < A2,R > => < A2',R' > .
  rl o < I1 + I2,R > => < I1 +Int I2,R > .
 crl o < A1 / A2,R > => < A1' / A2,R' > if o < A1,R > => < A1',R' > .
 crl o < A1 / A2,R > => < A1 / A2',R' > if o < A2,R > => < A2',R' > .
 crl o < I1 / I2,R > => < I1 /Int I2,R > if I2 =/=Bool 0 .
 crl o < A1 <= A2,R > => < A1' <= A2,R' > if o < A1,R > => < A1',R' > .
 crl o < I1 <= A2,R > => < I1 <= A2',R' > if o < A2,R > => < A2',R' > .
  rl o < I1 <= I2,R > => < I1 <=Int I2,R > .
 crl o < not B,R > => < not B',R' > if o < B,R > => < B',R' > .
  rl o < not  true,R > => < false,R > .
  rl o < not false,R > => <  true,R > .
 crl o < B1 and B2,R > => < B1' and B2,R' > if o < B1,R > => < B1',R' > .
  rl o < false and B2,R > => < false,R > .
  rl o <  true and B2,R > => < B2,R > .
 crl o < X := A,R > => < X := A',R' > if o < A,R > => < A',R' > .
 crl o < X := I,(state = Sigma, R) > => < skip,(state = Sigma[I / X], R) >
  if Sigma(X) =/=Bool undefined .
 crl o < S1 ; S2,R > => < S1' ; S2,R' > if o < S1,R > => < S1',R' > .
  rl o < skip ; S2,R > => < S2,R > .
 crl o < if B then S1 else S2,R > => < if B' then S1 else S2,R' > if o < B,R > => < B',R'  > .
  rl o < if  true then S1 else S2,R > => < S1,R > .
  rl o < if false then S1 else S2,R > => < S2,R > .
  rl o < while B do S,R > => < if B then (S ; while B do S) else skip,R > .
  rl o < var Xl ; S > => < S,(state = Xl |-> 0) > .
endm
```

Figure 3.27: The MSOS of IMP in Maude, including the definition of configurations.

3.6.4 Notes

Modular Structural Operational Semantics (MSOS) was introduced in 1999 by Mosses [60] and since then mainly developed by himself and his collaborators (e.g., [61, 62, 63]). In this section we used the implicitly-modular variant of MSOS introduced in [63], which, as acknowledged by the authors of [63], was partly inspired from discussions with us[5]. To be more precise, we used a slightly simplified version of implicitly-modular MSOS here. In MSOS in its full generality, one can also declare some transitions *unobservable*; to keep the presentation simpler, we here omitted all the observability aspects of MSOS.

The idea of our representation of MSOS into rewriting logic adopted in this section is taken over from Serbanuta *et al.* [85]. At our knowledge, Meseguer and Braga [50] give the first representation of MSOS into rewriting logic. The representation in [50] also led to the development of the Maude MSOS tool [18], which was the core of Braga's doctoral thesis. What is different in the representation of Meseguer and Braga in [50] from ours is that the former uses two different types of configuration wrappers, one for the left-hand side of the transitions and one for the right-hand side; this was already discussed in Section 3.3.4.

3.6.5 Exercises

Exercise 101. *Redo all the exercises in Section 3.3.5 but for the MSOS of* IMP *discussed in Section 3.6.1 instead of its small-step SOS in Section 3.3.2. Skip Exercises 52, 53 and 60, since the SOS proof system there drops the syntactic components in the RHS configurations in transitions, making it unsuitable for MSOS. For the MSOS variant of Exercise 54, just follow the same non-modular approach as in the case of SOS and not the modular MSOS approach discussed in Section 3.6.2 (Exercise 104 addresses the modular MSOS variant of abrupt termination).*

Exercise 102. *Same as Exercise 68, but for MSOS instead of small-step SOS: add variable increment to* IMP*, like in Section 3.6.2.*

Exercise 103. *Same as Exercise 72, but for MSOS instead of small-step SOS: add input/output to* IMP*, like in Section 3.6.2.*

Exercise 104. *Same as Exercise 77, but for MSOS instead of small-step SOS: add abrupt termination to* IMP*, like in Section 3.6.2.*

Exercise 105. *Same as Exercise 85, but for MSOS instead of small-step SOS: add dynamic threads to* IMP*, like in Section 3.6.2.*

Exercise 106. *Same as Exercise 90, but for MSOS instead of small-step SOS: add local variables using* `let` *to* IMP*, like in Section 3.6.2.*

Exercise 107. *This exercise asks to define* IMP++ *in MSOS, in various ways. Specifically, redo Exercises 95, 96, 97, 98, and 99, but for the MSOS of* IMP++ *discussed in Section 3.6.2 instead of its small-step SOS in Section 3.5.6.*

[5]In fact, drafts of this book that preceded [63] dropped the implicit labels in MSOS rules for notational simplicity; Mosses found that simple idea worthwhile formalizing within MSOS.