## 3.7 Reduction Semantics with Evaluation Contexts

The small-step SOS/MSOS approaches discussed in Sections 3.3 and 3.6 define a language semantics as a proof system whose rules are mostly conditional. The conditions of such rules allow to implicitly capture the program execution context as a *proof context*. This shift of focus from the informal notion of execution context to the formal notion of proof context has a series of advantages and it was, to a large extent, the actual point of formalizing language semantics using SOS. However, as the complexity of programming languages increased, in particular with the adoption of control-intensive statements like call/cc (see Chapter 12) that can arbitrarily change the execution context, the need for an explicit representation of the execution context as a first-class citizen in the language semantics also increased. Reduction semantics with evaluation contexts (RSEC) is a variant of small-step SOS where the evaluation context may appear explicit in the term being reduced.

In an RSEC language definition one starts by defining the syntax of *evaluation contexts*, or simply just *contexts*, which is typically done by means of a context-free grammar (CFG). A context is a program or a fragment of program with a *hole*, where the hole, which is written $\Box$, is a placeholder for where the next computational step can take place. If $c$ is an evaluation context and $e$ is some well-formed appropriate fragment (expression, statement, etc.), then $c[e]$ is the program or fragment obtained by replacing the hole of $c$ by $e$. Reduction semantics with evaluation contexts relies on a tacitly assumed (but rather advanced) parsing mechanism that takes a program or a fragment $p$ and decomposes it into a context $c$ and a subprogram or fragment $e$, called a *redex*, such that $p = c[e]$. This decomposition process is called *splitting* (of $p$ into $c$ and $e$). The inverse process, composing a redex $e$ and a context $c$ into a program or fragment $p$, is called *plugging* (of $e$ into $c$). These splitting/plugging operations are depicted in Figure 3.28.

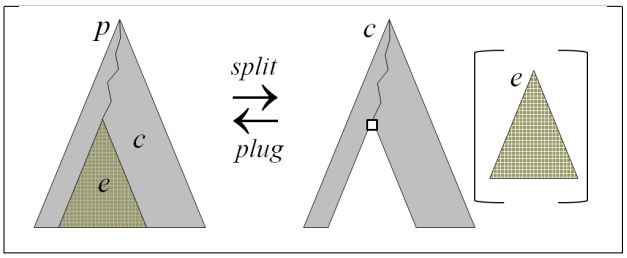


Figure 3.28: Decomposition of syntactic term $p$ into context $c$ and redex $e$, written $p = c[e]$: we say $p$ *splits* into $c$ and $e$, or $e$ *plugs* into $c$ yielding $p$. These operations are assumed whenever needed.

Consider a language with arithmetic/Boolean expressions and statements like our IMP language in Section 3.1. A possible CFG definition of evaluation contexts for such a language may include the following productions (the complete definition of IMP evaluation contexts is given in Figure 3.30):

$$
\begin{array}{rcl}
\mathit{Context} & ::= & \Box \\
& | & \mathit{Context} \texttt{ <= } \mathit{AExp} \\
& | & \mathit{Int} \texttt{ <= } \mathit{Context} \\
& | & \mathit{Id} \texttt{ := } \mathit{Context} \\
& | & \mathit{Context} \texttt{ ; } \mathit{Stmt} \\
& | & \texttt{if } \mathit{Context} \texttt{ then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} \\
& | & \ldots
\end{array}
$$

Note how the intended evaluation strategies of the various language constructs are reflected in the definition of evaluation contexts: `<=` is sequentially strict ($\Box$ allowed to go into the first subexpression until evaluated to an *Int*, then into the second subexpression), `:=` is strict only in its second argument while the sequential composition and the conditional are strict only in their first arguments. If one thinks of language constructs as operations taking a certain number

of arguments of certain types, then note that the operations appearing in the grammar defining evaluation contexts are *different* from their corresponding operations appearing in the language syntax; for example, $Id$ `:=` *Context* is different from $Id$ `:=` *AExp* because the former takes a context as second argument while the latter takes an arithmetic expression.

Here are some examples of correct evaluation contexts for the grammar above:

$\Box$

3 `<=` $\Box$

$\Box$ `<=` 3

$\Box$ `;` $x$ `:=`5, where $x$ is any variable

`if` $\Box$ `then` $s_1$ `else` $s_2$, where $s_1$ and $s_2$ are any well-formed statements

Here are some examples of incorrect evaluation contexts:

$\Box$ `<=` $\Box$ — a context can have only one hole

$x$ `<=` 3 — a context must contain a hole

$x$ `<=` $\Box$ — the first argument of `<=` must be an integer number in order to allow the hole in the second argument

$x$ `:=`5 `;` $\Box$ — the hole can only appear in the first statement in a sequential composition

$\Box$ `:=`5 — the hole cannot appear as first argument of `:=`

`if` $x$ `<=`7 `then` $\Box$ `else` $x$ `:=`5 — the hole is only allowed in the condition of a conditional

Here are some examples of decompositions of syntactic terms into a context and a redex (recall that in this book we can freely use parentheses for disambiguation; here we enclose evaluation contexts in parentheses for clarity):

$7 = (\Box)[7]$

$3 \texttt{<=} x = (3 \texttt{<=} \Box)[x] = (\Box \texttt{<=} x)[3] = (\Box)[3 \texttt{<=} x]$

$3 \texttt{<=} (2 \texttt{+} x) \texttt{+} 7 = (3 \texttt{<=} \Box \texttt{+} 7)[2 \texttt{+} x] = (\Box \texttt{<=} (2 \texttt{+} x) \texttt{+} 7)[3] = \ldots$

For simplicity, we consider only one type of context in this section, but in general one can have various types, depending upon the types of their holes and of their result.

Reduction semantics with evaluation contexts tends to be a purely syntactic definitional framework (following the slogan "everything is syntax"). If semantic components are necessary in a particular definition, then they are typically "swallowed by the syntax". For example, if one needs a state as part of the configuration for a particular language definition (like we need for our hypothetical IMP language discussed here), then one adds a context production of the form

$$\mathit{Context} \quad ::= \quad \langle \mathit{Context}, \mathit{State} \rangle$$

where the *State*, an inherently semantic entity, becomes part of the evaluation context. Note that once one adds additional syntax to evaluation contexts that does not correspond to constructs in the syntax of the original language, such as our pairing of a context and a state above, one needs to also extend the original syntax with corresponding constructs, so that the parsing-like mechanism decomposing a syntactic term into a context and a redex can be applied. In our case, the

production above suggests that a pairing configuration construct of the form $\langle \mathit{Stmt}, \mathit{State} \rangle$, like for SOS, also needs to be defined. Unlike in SOS, we do not need configurations pairing other syntactic categories with a state, such as $\langle \mathit{AExp}, \mathit{State} \rangle$ and $\langle \mathit{BExp}, \mathit{State} \rangle$; the reason is that, unlike in SOS, transitions with left-hand-side configurations $\langle \mathit{Stmt}, \mathit{State} \rangle$ are not derived anymore from transitions with left-hand-side configurations of the form $\langle \mathit{AExp}, \mathit{State} \rangle$ or $\langle \mathit{BExp}, \mathit{State} \rangle$.

Evaluation contexts are defined in such a way that whenever $e$ is reducible, $c[e]$ is also reducible. For example, consider the term $c[i_1 \texttt{<=} i_2]$ stating that expression $i_1 \texttt{<=} i_2$ is in a proper evaluation context. Since $i_1 \texttt{<=} i_2$ reduces to $i_1 \leq_{\mathit{Int}} i_2$, we can conclude that $c[i_1 \texttt{<=} i_2]$ reduces to $c[i_1 \leq_{\mathit{Int}} i_2]$. Therefore, in reduction semantics with evaluation contexts we can define the semantics of `<=` using the following rule:

$$c[i_1 \texttt{<=} i_2] \to c[i_1 \leq_{\mathit{Int}} i_2]$$

This rule is actually a rule schema, containing one rule instance for each concrete integers $i_1, i_2$ and for each appropriate evaluation context $c$. For example, here are instances of this rule when the context is $\Box$, $\texttt{if } \Box \texttt{ then skip else } x \texttt{:=} 5$, and $\langle \Box, (x \mapsto 1, y \mapsto 2) \rangle$, respectively:

$$\begin{gathered} i_1 \texttt{<=} i_2 \to i_1 \leq_{\mathit{Int}} i_2 \\ \texttt{if } (i_1 \texttt{<=} i_2) \texttt{ then skip else } x \texttt{:=} 5 \to \texttt{if } (i_1 \leq_{\mathit{Int}} i_2) \texttt{ then skip else } x \texttt{:=} 5 \\ \langle i_1 \texttt{<=} i_2 \ , \ (x \mapsto 1, y \mapsto 2) \rangle \to \langle i_1 \leq_{\mathit{Int}} i_2 \ , \ (x \mapsto 1, y \mapsto 2) \rangle \end{gathered}$$

What is important to note here is that propagation rules, such as (MSOS-Leq-Arg1) and (MSOS-Leq-Arg2) in Figure 3.23, are not necessary anymore when using evaluation contexts, because the evaluation contexts already achieve the role of the propagation rules.

To reflect the fact that reductions take place only in appropriate contexts, RSEC typically introduces a rule schema of the form:

$$\frac{e \to e'}{c[e] \to c[e']} \qquad \text{(RSEC-Characteristic-Rule)}$$

where $e, e'$ are well-formed fragments and $c$ is any appropriate evaluation context (i.e., such that $c[e]$ and $c[e']$ are well-formed programs or fragments of program). This rule is called the *characteristic rule* of RSEC. When this rule is applied, we say that $e$ *reduces to* $e'$ *in context* $c$. If one picks $c$ to be the empty context $\Box$, then $c[e]$ is $e$ and thus the characteristic rule is useless; for that reason, the characteristic rule may be encountered with a side condition "if $c \neq \Box$". Choosing good strategies to search for splits of terms into contextual representations can be a key factor in obtaining efficient implementations of RSEC execution engines.

The introduction of the characteristic rule allows us to define reduction semantics of languages or calculi quite compactly. For example, here are all the rules needed to completely define the semantics of the comparison ( `<=` ), sequential composition ( `;` ) and conditional ( `if` ) language constructs for which we defined evaluation contexts above:

$$\begin{aligned} i_1 \texttt{<=} i_2 &\to i_1 \leq_{\mathit{Int}} i_2 \\ \texttt{skip ; } s_2 &\to s_2 \\ \texttt{if true then } s_1 \texttt{ else } s_2 &\to s_1 \\ \texttt{if false then } s_1 \texttt{ else } s_2 &\to s_2 \end{aligned}$$

The characteristic rule tends to be the only conditional rule in an RSEC, in the sense that the remaining rules take no reduction premises (though they may still have side conditions). Moreover,

as already pointed out, the characteristic rule is actually unnecessary, because one can very well replace each rule $l \to r$ by a rule $c[l] \to c[r]$. The essence of reduction semantics with evaluation contexts is not its characteristic reduction rule, but its specific approach to defining evaluation contexts as a grammar and then using them as an explicit part of languages or calculi definitions. The characteristic reduction rule can therefore be regarded as "syntactic sugar", or convenience to the designer allowing her to write more compact definitions.

To give the semantics of certain language constructs, one may need to access specific information that is stored inside an evaluation context. For example, consider a term $\langle x \texttt{<=} 3 \ , \ (x \mapsto 1, y \mapsto 2)\rangle$, which can be split as $c[x]$, where $c$ is the context $\langle \Box \texttt{<=} 3 \ , \ (x \mapsto 1, y \mapsto 2)\rangle$. In order to reduce $c[x]$ to $c[1]$ as desired, we need to look inside $c$ and find out that the value of $x$ in the state held by $c$ is 1. Therefore, following the purely syntactic style adopted so far in this section, the reduction semantics with evaluation contexts rule for variable lookup in our case here is the following:

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

Indeed, the same way we add as much structure as needed in ordinary terms, we can add as much structure as needed in evaluation contexts. Similarly, below is the rule for variable assignment:

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

Note that in this case both the context and the redex were changed by the rule. In fact, as discussed in Section 3.10, one of the major benefits of reduction semantics with evaluation contexts consists in precisely the fact that one can arbitrarily modify the evaluation context in rules; this is crucial for giving semantics to control-intensive language constructs such as call/cc.

Splitting of a term into an evaluation context and a redex does not necessarily need to take place at the top of the left-hand side of a rule. For example, the following is an alternative way to give reduction semantics with evaluation contexts to variable lookup and assignment:

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

Note that, even if one decides to follow this alternative style, one still needs to include the production $\mathit{Context} ::= \langle \mathit{Context}, \mathit{State}\rangle$ to the evaluation context CFG if one wants to write rules as $c[i_1 \texttt{<=} i_2] \to c[i_1 \leq_{\mathit{Int}} i_2]$ or to further take advantage of the characteristic rule and write elegant and compact rules such as $i_1 \texttt{<=} i_2 \to i_1 \leq_{\mathit{Int}} i_2$. If one wants to completely drop evaluation context productions that mix syntactic and semantic components, such as $\mathit{Context} ::= \langle \mathit{Context}, \mathit{State}\rangle$, then one may adopt one of the styles discussed in Exercises 108 and 109, respectively, though one should be aware of the fact that those styles also have their disadvantages.

Figure 3.29 shows a reduction sequence using the evaluation contexts and the rules discussed so far. In Figure 3.29, we used the following (rather standard) notation for instantiated contexts whenever we applied the characteristic rule: the redex is placed in a box replacing the hole of the context. For example, the fact that expression $3 \texttt{<=} x$ is split into contextual representation $(3 \texttt{<=} \Box)[x]$ is written compactly and intuitively as $3 \texttt{<=} \boxed{x}$. Note that the evaluation context changes almost at each step during the reduction sequence in Figure 3.29.

Like in small-step SOS and MSOS, we can also transitively and reflexively close the one-step transition relation $\to$. As usual, we let $\to^\star$ denote the resulting multi-step transition relation.

$$
\begin{array}{ll}
 & \langle \boxed{x := 1}\ ;\ y := 2\ ;\ \texttt{if}\ x <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 0, y \mapsto 0)\rangle \\
\rightarrow & \langle \boxed{\texttt{skip}\ ;\ y := 2}\ ;\ \texttt{if}\ x <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 0)\rangle \\
\rightarrow & \langle \boxed{y := 2}\ ;\ \texttt{if}\ x <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 0)\rangle \\
\rightarrow & \langle \texttt{skip}\ ;\ \texttt{if}\ x <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\ \boxed{x}\ <= y\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\ 1 <= \boxed{y}\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if}\ \boxed{1 <= 2}\ \texttt{then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{if true then}\ x := 0\ \texttt{else}\ y := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle x := 0\ ,\ (x \mapsto 1, y \mapsto 2)\rangle \\
\rightarrow & \langle \texttt{skip}\ ,\ (x \mapsto 0, y \mapsto 2)\rangle
\end{array}
$$

Figure 3.29: Sample reduction sequence.

| IMP evaluation contexts syntax | IMP language syntax |
|---|---|
| *Context* ::= □<br>\| *Context* + *AExp* \| *AExp* + *Context*<br>\| *Context* / *AExp* \| *AExp* / *Context*<br><br>\| *Context* <= *AExp* \| *Int* <= *Cxt*<br>\| not *Context*<br>\| *Context* and *BExp*<br>\| *Id* := *Context*<br>\| *Context* ; *Stmt*<br>\| if *Context* then *Stmt* else *Stmt* | *AExp* ::= *Int* \| *Id* \|<br>\| *AExp* + *AExp*<br>\| *AExp* / *AExp*<br>*BExp* ::= *Bool*<br>\| *AExp* <= *AExp*<br>\| not *BExp*<br>\| *BExp* and *BExp*<br>*Stmt* ::= *Id* := *AExp*<br>\| *Stmt* ; *Stmt*<br>\| if *BExp* then *Stmt* else *Stmt*<br>\| while *BExp* do *Stmt*<br>*Pgm* ::= var **List**{*Id*} ; *Stmt* |

Figure 3.30: Evaluation contexts for IMP (left column); the syntax of IMP (from Figure 3.1) is recalled in the right column only for reader's convenience, to more easily compare the two grammars.

### 3.7.1 The Reduction Semantics with Evaluation Contexts of IMP

Figure 3.30 shows the definition of evaluation contexts for IMP and Figure 3.31 shows all the reduction semantics rules of IMP using the evaluation contexts defined in Figure 3.30. The evaluation context productions capture the intended evaluation strategies of the various language constructs. For example, `+` and `/` are non-deterministically strict, so any one of their arguments can be reduced one step whenever the sum or the division expression can be reduced one step, respectively, so the hole $\square$ can go in any of their two subexpressions. As previously discussed, in the case of `<=` one can reduce its second argument only after its first argument is fully reduced (to an integer). The evaluation strategy of `not` is straightforward. For `and`, note that only its first argument is reduced. Indeed, recall that `and` has a short-circuited semantics, so its second argument is reduced only after the first one is completely reduced (to a Boolean) and only if needed; this is defined using rules in Figure 3.31. The evaluation contexts for assignment, sequential composition, and the conditional have already been discussed.

Many of the rules in Figure 3.31 have already been discussed or are trivial and thus deserve no discussion. Note that there is no production *Context* ::= `while` *Context* `do` *Stmt* as a hasty reader may (mistakenly) expect. That is because such a production would allow the evaluation of the Boolean expression in the while loop's condition to a Boolean value in the current context; supposing that value is `true`, then, unless one modifies the syntax in some rather awkward way, there is no chance to recover the original Boolean expression to evaluate it again after the evaluation of the while loop's body statement. The solution to handle loops remains the same as in SOS, namely to explicitly unroll them into conditional statements, as shown in Figure 3.31. Note that the evaluation contexts allow the loop unrolling to only happen when the `while` statement is a redex. In particular, after an unrolling reduction takes place, subsequent unrolling steps are disallowed inside the `then` branch; to unroll it again, the loop statement must become again a redex, which can only happen after the conditional statement is itself reduced. The initial program configuration (containing only the program) is reduced also like in SOS (last rule in Figure 3.31; note that one cannot instead define a production *Context* ::= `var` **List**`{`*Id*`}` `;` *Context*, because, for example, there is no way to reduce the statement $x$ `:=` $5$ in $\langle$ `var` $x$ `;` $\boxed{x \texttt{ := } 5}\rangle$.

### 3.7.2 The Reduction Semantics with Evaluation Contexts of IMP++

We next discuss the reduction semantics of IMP++ using evaluation contexts. Like for the other semantics, we first add each feature separately to IMP and then we add all of them together and investigate the modularity and appropriateness of the resulting definition.

#### Variable Increment

The use of evaluation contexts makes the definition of variable increment quite elegant and modular:

$$\langle c, \sigma\rangle[\texttt{++}\, x] \quad \rightarrow \quad \langle c, \sigma[(\sigma(x) +_{Int} 1)/x]\rangle[\sigma(x) +_{Int} 1]$$

No other rule needs to change, because: (1) unlike in SOS, all the language-specific rules are unconditional, each rule matching and modifying only its relevant part of the configuration; and (2) the language-independent characteristic rule allows reductions to match and modify only their relevant part of the configuration, propagating everything else in the configuration automatically.

$$Context ::= \ldots \mid \langle Context, State \rangle$$

$$\frac{e \to e'}{c[e] \to c[e']}$$

$$\begin{array}{rcll}
\langle c, \sigma \rangle[x] & \to & \langle c, \sigma \rangle[\sigma(x)] & \text{if } \sigma(x) \neq \bot \\
i_1 \texttt{ + } i_2 & \to & i_1 +_{Int} i_2 & \\
i_1 \texttt{ / } i_2 & \to & i_1 /_{Int} i_2 & \text{if } i_2 \neq 0 \\
i_1 \texttt{ <= } i_2 & \to & i_1 \leq_{Int} i_2 & \\
\texttt{not true} & \to & \texttt{false} & \\
\texttt{not false} & \to & \texttt{true} & \\
\texttt{true and } b_2 & \to & b_2 & \\
\texttt{false and } b_2 & \to & \texttt{false} & \\
\langle c, \sigma \rangle[x \texttt{ := } i] & \to & \langle c, \sigma[i/x] \rangle[\texttt{ skip }] & \text{if } \sigma(x) \neq \bot \\
\texttt{skip ; } s_2 & \to & s_2 & \\
\texttt{if true then } s_1 \texttt{ else } s_2 & \to & s_1 & \\
\texttt{if false then } s_1 \texttt{ else } s_2 & \to & s_2 & \\
\texttt{while } b \texttt{ do } s & \to & \texttt{if } b \texttt{ then } (s \texttt{ ; while } b \texttt{ do } s) \texttt{ else skip} & \\
\langle \texttt{ var } xl \texttt{ ; } s \rangle & \to & \langle s, (xl \mapsto 0) \rangle &
\end{array}$$

Figure 3.31: RSEC(IMP): The reduction semantics with evaluation contexts of IMP ($e, e' \in AExp \cup BExp \cup Stmt$; $c \in Context$ appropriate (that is, the respective terms involving $c$ are well-formed); $i, i_1, i_2 \in Int$; $x \in Id$; $b, b_2 \in BExp$; $s, s_1, s_2 \in Stmt$; $xl \in \mathbf{List}\{Id\}$; $\sigma \in State$).

### Input/Output

We need to first change the configuration employed by our reduction semantics with evaluation contexts of IMP from $\langle s, \sigma \rangle$ to $\langle s, \sigma, \omega_{in}, \omega_{out} \rangle$, to also include the input/output buffers. This change, unfortunately, generates several other changes in the existing semantics, some of them non-modular in nature. First, we need to change the syntax of (statement) configurations to include input and output buffers, and the syntax of contexts from $Context ::= \ldots \mid \langle Context, State \rangle$ to $Context ::= \ldots \mid \langle Context, State, Buffer, Buffer \rangle$. No matter what semantic approach one employs, some changes in the configuration (or its equivalent) are unavoidable when one adds new language features that require new semantic data, like input/output constructs that require their own buffers (recall that, e.g., in MSOS, we had to add new attributes in transition labels instead). Hence, this change is acceptable. What is inconvenient (and non-modular), however, is that the rules for variable lookup and for assignment need the complete configuration, so they have to change from

$$\begin{array}{rcl}
\langle c, \sigma \rangle[x] & \to & \langle c, \sigma \rangle[\sigma(x)] \\
\langle c, \sigma \rangle[x \texttt{ := } i] & \to & \langle c, \sigma[i/x] \rangle[\texttt{skip}]
\end{array}$$

to

$$\begin{array}{rcl}
\langle c, \sigma, \omega_{in}, \omega_{out} \rangle[x] & \to & \langle c, \sigma, \omega_{in}, \omega_{out} \rangle[\sigma(x)] \\
\langle c, \sigma, \omega_{in}, \omega_{out} \rangle[x \texttt{ := } i] & \to & \langle c, \sigma[i/x], \omega_{in}, \omega_{out} \rangle[\texttt{skip}]
\end{array}$$

Also, the initial configuration which previously held only the program, now has to change to hold both the program and an input buffer, and the rule for programs (the last one in Figure 3.31) needs

to change as follows:

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

Once the changes above are applied, we are ready for adding the evaluation context for the print statement as well as the reduction semantics rules of both input/output constructs:

$$\begin{array}{rcl} \mathit{Context} & ::= & \ldots \mid \texttt{print(}\, \mathit{Context} \texttt{)} \\ \langle c, \sigma, i : \omega_{in}, \omega_{out} \rangle [\texttt{read()}] & \rightarrow & \langle c, \sigma, \omega_{in}, \omega_{out} \rangle [i] \\ \langle c, \sigma, \omega_{in}, \omega_{out} \rangle [\texttt{print(}\, i \texttt{)}] & \rightarrow & \langle c, \sigma, \omega_{in}, \omega_{out} : i \rangle [\texttt{skip}] \end{array}$$

Other possibilities to add input/output buffers to the configuration and to give the reduction semantics with evaluation contexts of the above language features in a way that appears to be more modular (but which yields other problems) are discussed in Section 3.10.

### Abrupt Termination

For our language, reduction semantics allows very elegant, natural and modular definitions of abrupt termination, without having to extent the syntax of the original language and without adding any new reduction steps as an artifact of the approach chosen:

$$\begin{array}{rcl} \langle c, \sigma \rangle [i \,/\, 0] & \rightarrow & \langle\, \texttt{skip}\, , \sigma \rangle \\ \langle c, \sigma \rangle [\,\texttt{halt}\,] & \rightarrow & \langle\, \texttt{skip}\, , \sigma \rangle \end{array}$$

Therefore, the particular evaluation context in which the abrupt termination is being generated, $c$, is simply discarded. This is not possible in any of the big-step, small-step or MSOS styles above, because in those styles the evaluation context $c$ is captured by the proof context, which, like in any logical system, cannot be simply discarded. The elegance of the two rules above suggest that having the possibility to explicitly match and change the evaluation context is a very powerful and convenient feature of a language semantic framework.

### Dynamic Threads

The rules (SmallStep-Spawn-Arg) (resp. (MSOS-Spawn-Arg)) and (SmallStep-Spawn-Wait) (resp. (MSOS-Spawn-Wait)) in Section 3.5.4 (resp. Section 3.6.2) are essentially computation propagation rules. In reduction semantics with evaluation contexts the role of such rules is taken over by the splitting/plugging mechanism, which in turn relies on parsing and therefore needs productions for evaluation contexts. We can therefore replace those rules by appropriate productions for evaluation contexts:

$$\begin{array}{rcl} \mathit{Context} & ::= & \ldots \\ & \mid & \texttt{spawn(}\, \mathit{Context} \texttt{)} \\ & \mid & \texttt{spawn(}\, \mathit{Stmt} \texttt{)}\, \texttt{;}\, \mathit{Context} \end{array}$$

The second evaluation context production above involves two language constructs, namely `spawn` and sequential composition. The desired non-determinism due to concurrency is captured by deliberate ambiguity in parsing evaluation contexts and, implicitly, in the splitting/plugging mechanism.

The remaining rule (SmallStep-Spawn-Skip) (resp. (MSOS-Spawn-Skip)) in Section 3.5.4 (resp. Section 3.6.2) is turned into an equivalent rule here, and the structural identity stating the associativity of sequential composition is also still necessary:

$$\begin{array}{rcl} \texttt{spawn skip} & \rightarrow & \texttt{skip} \\ (s_1\, \texttt{;}\, s_2)\, \texttt{;}\, s_3 & \equiv & s_1\, \texttt{;}\, (s_2\, \texttt{;}\, s_3) \end{array}$$

### Local Variables

We make use of the procedure presented in Section 3.5.5 for desugaring blocks with local variables into `let` constructs, to reduce the problem to only give semantics to `let`. Recall from Section 3.5.5 that the semantics of $\texttt{let}\, x = a\, \texttt{in}\, s$ in a state $\sigma$ is to first evaluate arithmetic expression $a$ in $\sigma$ to some integer $i$ and then evaluate statement $s$ in state $\sigma[i/x]$; after the evaluation of $s$, the value of $x$ is recovered to $\sigma(x)$ (i.e., whatever it was before the execution of the block) but all the other state updates produced by the evaluation of $s$ are kept. These suggest the following:

$$\textit{Context} \;::=\; \ldots \;\mid\; \texttt{let}\; \textit{Id} = \textit{Context}\; \texttt{in}\; \textit{Stmt}$$

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

Notice that a solution similar to that in small-step SOS and MSOS (see rules (SmallStep-Let-Stmt) and (MSOS-Let-Stmt) in Sections 3.5.5 and 3.6.2, respectively) does not work here, because rules in reduction semantics with evaluation contexts are unconditional. In fact, a solution similar to the one we adopted above was already discussed in Section 3.5.6 in the context of small-step SOS, where we also explained that it works as shown because of a syntactic trick, namely because we allow assignment statements of the form $x := \bot$ (in our case here when $\sigma(x) = \bot$), which have the effect to undefine $\sigma$ in $x$ (see Section 2.3.2). Section 3.5.6 also gives suggestions on how to avoid allowing $\bot$ to be assigned to $x$, if one does not like it. One suggestion was to rename the bound variable into a fresh one, this way relieving us from having to recover its value after the `let`:

$$\langle c, \sigma\rangle[\texttt{let}\, x = i\, \texttt{in}\, s] \;\rightarrow\; \langle c, \sigma[i/x']\rangle[s[x'/x]] \qquad \text{where } x' \text{ fresh}$$

This approach comes with several problems, though: it requires that we define and maintain a substitution operation (for $s[x'/x]$), we have to pay a complexity linear with the size of the `let` body each time a `let` statement is eliminated, and the state may grow indefinitely (since the `let` can be inside a loop).

Note also that, with the current reduction semantics with evaluation contexts of IMP, we cannot add the following evaluation context production

$$\textit{Context} \;::=\; \ldots \;\mid\; \texttt{let}\; \textit{Id} = \textit{Int}\; \texttt{in}\; \textit{Context}$$

stating that once the binding expression becomes an integer then we can evaluate the `let` body. We cannot add it simply because we have to bind the variable to the integer before we evaluate the `let` body statement. An evaluation context production like above would result in evaluating the `let` body statement in the same state as before the `let`, which is obviously wrong. However, the existence of a `let` binder allows us to possibly rethink the overall reduction semantics of IMP, to make it more syntactic. Indeed, the `let` binders can be used in a nested manner and hereby allow us to syntactically mimic a state. For example, a statement of the form $\texttt{let}\, x = 5\, \texttt{in}\, \texttt{let}\, y = 7\, \texttt{in}\, s$ can be thought of as the statement $s$ being executed in a "state" where $x$ is bound to 5 and where $y$ is bound to 7. We can then drop the configurations of IMP completely and instead add the evaluation context above allowing reductions inside `let` body statements. The IMP rules that refer to configurations, namely those for lookup and assignment, need to change to work with the new "state", and a new rule to eliminate unnecessary `let` statements needs to be added:

$$\begin{array}{rcl} \texttt{let}\, x = i\, \texttt{in}\, c[x] & \rightarrow & \texttt{let}\, x = i\, \texttt{in}\, c[i] \\ \texttt{let}\, x = i\, \texttt{in}\, c[x := j] & \rightarrow & \texttt{let}\, x = j\, \texttt{in}\, c[\texttt{skip}] \\ \texttt{let}\, x = i\, \texttt{in}\, \texttt{skip} & \rightarrow & \texttt{skip} \end{array}$$

The above works correctly only if one ensures that the evaluation contexts $c$ do not contain other $\texttt{let}\, x = \_\, \texttt{in}\, \_$ evaluation context constructs, with the same $x$ variable name as in the rules (the underscores can be any integer and context, respectively). One can do this by statically renaming the bound variables to have different names at parse-time, or by employing a substitution operation to do it dynamically. Note that the static renaming approach requires extra-care as the language is extended, particularly if new `let` statements can be generated dynamically by other language constructs. Another approach to ensure the correct application of the rules above, which is theoretically more complex and practically more expensive and harder to implement, is to add a side condition to the first two rules of the form "where $c$ does not contain any evaluation context production instance of the form $\texttt{let}\, x = \_\, \texttt{in}\, \_$".

To conclude, the discussion above suggests that there are various ways to give a reduction semantics of `let` using evaluation contexts, none of them absolutely better than the others: some are simpler, others are more syntactic but require special external support, others require non-modular changes to the existing language. The approaches above are by no means exhaustive. For example, we have not even discussed environment-store based approaches.

### Putting Them All Together

It is relatively easy to combine all the reduction semantics with evaluation contexts of all the features above, although not as modularly as it was for MSOS. Specifically, we have to do the following:

1. Apply all the changes that we applied when we added input/output to IMP above, namely: add input/output buffers to both configurations and configuration evaluation contexts; change the semantic rules involving configurations to work with the extended configurations, more precisely the rules for variable lookup, for variable assignment, and for programs.

2. Add all the evaluation contexts and the rules for the individual features above, making sure we *change* those of them using configurations or configuration evaluation contexts (i.e., almost all of them) to work with the new configurations including input/output buffers.

Unfortunately, the above is not giving us the desired language. Worse, it actually gives us a wrong language, namely one with a disastrous feature interaction. This problem has already been noted in Section 3.5.6, where we discussed the effect of a similar semantics to that of `let` above, but using small-step SOS instead of evaluation contexts. The problem here is that, if the body of a `let` statement contains a `spawn` statement, then the latter will be allowed, according to its semantics, to be executed in parallel with the statements following it. In our case, the assignment statement $x := \sigma(x)$ in the `let` semantics, originally intended to recover the value of $x$, can be now potentially executed before the spawned statement, resulting in a wrong behavior; in particular, the assignment can even "undefine" $x$ in case $\sigma(x) = \bot$, in which case the `spawn` statement can even get stuck.

As already indicated in Section 3.5.6, the correct way to eliminate the `let` construct is to rename the bound variable into a fresh variable visible only to `let`'s body statement, this way eliminating the need to recover the value of the bound variable to what it was before the `let`:

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

We have used configurations already extended with input/output buffers, as needed for IMP++. This solution completely brakes any (intended or unintended) relationship between the `let` construct and any other language constructs that may be used inside its body, although, as discussed in Section 3.5.6, the use of the substitution comes with a few (relatively acceptable) drawbacks.

**sort:**
$Syntax$ // includes all syntactic terms, in contextual representation $context[redex]$ or not
**subsorts:**
$N_1, N_2, \ldots < Syntax$ // $N_1$, $N_2$, ..., are sorts whose terms can be regarded as $context[redex]$
**operations:**
$\_[\_] : Context \times Syntax \to Syntax$ // constructor for terms in contextual representation
$split : Syntax \to Syntax$ // puts syntactic terms into contextual representation
$plug : Syntax \to Syntax$ // the dual of split
**rules and equations:**
$split(Syn) \to \Box[Syn]$ // generic rule; it initiates the splitting process for the rules below
$plug(\Box[Syn]) = Syn$ // generic equation; it terminates the plugging process
// for each context production $Context ::= \pi(N_1, \ldots, N_n, Context)$ add the following:
$split(\pi(T_1, \ldots, T_n, T)) \to \pi(T_1, \ldots, T_n, C)[Syn]$ **if** $split(T) \to C[Syn]$
$plug(\pi(T_1, \ldots, T_n, C)[Syn]) = \pi(T_1, \ldots, T_n, plug(C[Syn]))$

Figure 3.32: Embedding evaluation contexts into rewriting logic theory $\mathcal{R}^{\Box}_{\mathrm{RSEC}}$. The implicit split/plug mechanism is replaced by explicit rewriting logic sentences achieving the same task (the involved variables have the sorts $Syn : Syntax$, $C : Context$, $T_1 : N_1$, ..., $T_n : N_n$, and $T : N$).

### 3.7.3 Reduction Semantics with Evaluation Contexts in Rewriting Logic

In this section we show how to automatically and faithfully embed reduction semantics with evaluation contexts into rewriting logic. After discussing how to embed evaluation contexts into rewriting logic, we first give a straightforward embedding of reduction semantics, which is easy to prove correct but which does not take advantage of performance-improving techniques currently supported by rewrite engines, so consequently it is relatively inefficient when executed or formally analyzed. We then discuss simple optimizations which increase the performance of the resulting rewrite definitions an order of magnitude or more. We only consider evaluation contexts which can be defined by means of context-free grammars (CFGs). However, the CFG that we allow for defining evaluation contexts can be non-deterministic, in the sense that a term is allowed to split many different ways into a context and a redex (like the CFG in Figure 3.30).

#### Faithful Embedding of Evaluation Contexts into Rewriting Logic

Our approach to embedding reduction semantics with evaluation contexts in rewriting logic builds on an embedding of evaluation contexts and their implicit splitting/plugging mechanism in rewriting logic. More precisely, each evaluation context production is associated with an equation (for plugging) and a conditional rewrite rule (for splitting). The conditional rewrite rules allow to non-deterministically split a term into a context and a redex. Moreover, when executing the resulting rewriting logic theory, the conditional rules allow for finding *all* splits of a term into a context and a redex, provided that the underlying rewrite engine has search capabilities (like Maude does).

Figure 3.32 shows a general and automatic procedure to generate a rewriting logic theory from any CFG defining evaluation contexts for some given language syntax. Recall that, for simplicity, in this section we assume only one *Context* syntactic category. What links the CFG of evaluation contexts to the CFG of the language to be given a semantics, which is also what makes our embedding

into rewriting logic discussed here work, is the assumption that for any context production

$$Context \ ::= \ \pi(N_1, \ldots, N_n, Context)$$

there are some syntactic categories $N, N'$ (different or not) in the language CFG (possibly extended with configurations and semantic components as discussed above) such that $\pi(t_1, \ldots, t_n, t) \in N'$ for any $t_1 \in N_1$, ..., $t_n \in N_n$, $t \in N$. We here used a notation which needs to be explained. The actual production above is $Context ::= \pi$, where $\pi$ is a string of terminals and non-terminals, but we write $\pi(N_1, \ldots, N_n, Context)$ instead of $\pi$ to emphasize that $N_1, \ldots, N_n, Context$ are all the non-terminals appearing in $\pi$; we listed the *Context* last for simplicity. Also, by abuse of notation, we let $\pi(t_1, \ldots, t_n, t)$ denote the term obtained by substituting $(t_1, \ldots, t_n, t)$ for $(N_1, \ldots, N_n, Context)$ in $\pi$, respectively. So our assumption is that $\pi(t_1, \ldots, t_n, t)$ is well-formed under the syntax of the language whenever $t_1$, ..., $t_n$, $t$ are well-defined in the appropriate syntactic categories, that is, $t_1 \in N_1$, ..., $t_n \in N_n$, $t \in T$. This is indeed a very natural property of well-defined evaluation contexts for a given language, so natural that one may even ask how it can be otherwise (it is easy to violate this property though, e.g., $Context ::= \texttt{<=} Context \texttt{:=} AExp$). Without this property, our embedding of evaluation contexts in rewriting logic in Figure 3.32 would not be well-formed, because the left-hand side terms of some of the conditional rule(s) for split would not be well-formed terms.

For simplicity, in Figure 3.32 we prefer to subsort all the syntactic categories whose terms are intended to be allowed contextual representations $context[redex]$ under one top sort[6], *Syntax*. The implicit notation $context[term]$ for contextual representations, as well as the implicitly assumed *split* and *plug* operations, are defined explicitly in the corresponding rewrite theory. The split operation is only defined on terms over the original language syntax, while the plug operation is defined only over terms in contextual representation. One generic (i.e., independent of the particular RSEC definition) rule and one generic equation are added: $split(Syn) \rightarrow \Box[Syn]$ initiates the process of splitting a term into a contextual representation and $plug(\Box[Syn]) = Syn$ terminates the process of plugging a term into a context. It is important that the first be a rewrite rule (because it can lead to non-determinism; this is explained below), while the second can safely be an equation.

Each evaluation context production translates into one equation and one conditional rewrite rule. The equation tells how terms are plugged into contexts formed with that production, while the conditional rule tells how that production can be used to split a term into a context and a redex. The equations defining plugging are straightforward: for each production in the original CFG of evaluation contexts, iteratively plug the subterm in the smaller context; when the hole is reached, replace it by the subterm via the generic equation. The conditional rules for splitting also look straightforward, but how and why they work is more subtle. For any context production, if the term to split matches the pattern of the production, then first split the subterm corresponding to the position of the subcontext and then use that contextual representation of the subterm to construct the contextual representation of the original term; at any moment, one has the option to stop splitting thanks to the generic rule $split(Syn) \rightarrow \Box[Syn]$. For example, for the five *Context*

[6] An alternative, which does not involve subsorting, is to replace each syntactic category as above by *Syntax*. Our construction also works without subsorting and without collapsing of syntactic categories, but it is more technical, requires more operations, rules, and equations, and it is likely not worth the effort without a real motivation to use it in term rewrite settings without support for subsorting. We have made experiments with both approaches and found no penalty on performance when collapsing syntactic categories.

productions in the evaluation context CFG in the preamble of this section, namely

$$\begin{array}{rcl} \mathit{Context} & ::= & \mathit{Context} \texttt{ <= } \mathit{AExp} \\ & | & \mathit{Int} \texttt{ <= } \mathit{Context} \\ & | & \mathit{Id} \texttt{ := } \mathit{Context} \\ & | & \mathit{Context} \texttt{ ; } \mathit{Stmt} \\ & | & \texttt{if } \mathit{Context} \texttt{ then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} \end{array}$$

the general procedure in the rewriting logic embedding of evaluation contexts in Figure 3.32 yields the following five rules and five equations (variable $I_1$ has sort *Int*; $X$ has sort *Id*; $A_1$ and $A_2$ have sort *AExp*; $B$ has sort *BExp*; $S_1$ and $S_2$ have sort *Stmt*; $C$ has sort *Context*; *Syn* has sort *Syntax*):

$$\begin{array}{l} \mathit{split}(A_1 \texttt{ <= } A_2) \to (C \texttt{ <= } A_2)[\mathit{Syn}] \textbf{ if } \mathit{split}(A_1) \to C[\mathit{Syn}] \\ \mathit{plug}((C \texttt{ <= } A_2)[\mathit{Syn}]) = \mathit{plug}(C[\mathit{Syn}]) \texttt{ <= } A_2 \end{array}$$

$$\begin{array}{l} \mathit{split}(I_1 \texttt{ <= } A_2) \to (I_1 \texttt{ <= } C)[\mathit{Syn}] \textbf{ if } \mathit{split}(A_2) \to C[\mathit{Syn}] \\ \mathit{plug}((I_1 \texttt{ <= } C)[\mathit{Syn}]) = I_1 \texttt{ <= } \mathit{plug}(C[\mathit{Syn}]) \end{array}$$

$$\begin{array}{l} \mathit{split}(X \texttt{ := } A) \to (X \texttt{ := } C)[\mathit{Syn}] \textbf{ if } \mathit{split}(A) \to C[\mathit{Syn}] \\ \mathit{plug}((X \texttt{ := } C)[\mathit{Syn}]) = X \texttt{ := } \mathit{plug}(C[\mathit{Syn}]) \end{array}$$

$$\begin{array}{l} \mathit{split}(S_1 \texttt{ ; } S_2) \to (C \texttt{ ; } S_2)[\mathit{Syn}] \textbf{ if } \mathit{split}(S_1) \to C[\mathit{Syn}] \\ \mathit{plug}((C \texttt{ ; } S_2)[\mathit{Syn}]) = \mathit{plug}(C[\mathit{Syn}]) \texttt{ ; } S_2 \end{array}$$

$$\begin{array}{l} \mathit{split}(\texttt{ if } B \texttt{ then } S_1 \texttt{ else } S_2) \to (\texttt{ if } C \texttt{ then } S_1 \texttt{ else } S_2)[\mathit{Syn}] \textbf{ if } \mathit{split}(B) \to C[\mathit{Syn}] \\ \mathit{plug}((\texttt{ if } C \texttt{ then } S_1 \texttt{ else } S_2)[\mathit{Syn}]) = \texttt{ if } \mathit{plug}(C[\mathit{Syn}]) \texttt{ then } S_1 \texttt{ else } S_2 \end{array}$$

The reason for which we used rules instead of equations for splitting in our embedding in Figure 3.32 is that splitting, unlike plugging, can be non-deterministic. Recall that the use of an arrow/transition in the condition of a rule has an existential nature (Sections 2.7). In particular, rewriting logic engines should execute conditional rules by performing an exhaustive search, or reachability analysis of all the zero-, one- or more-step rewrites of the condition left-hand side term ($\mathit{split}(T)$ in our case) into the condition right-hand side term ($C[\mathit{Syn}]$ in our case). Such an exhaustive search explores all possible splits, or parsings, of a term into a contextual representation.

**Theorem 6.** ***(Embedding splitting/plugging into rewriting logic)*** *Given an evaluation context CFG as discussed above, say as part of some reduction semantics with evaluation contexts definition* RSEC*, let* $\mathcal{R}^{\square}_{\text{RSEC}}$ *be the rewriting logic theory associated to it as in Figure 3.32. Then the following are equivalent for any* $t, r \in \mathit{Syntax}$ *and* $c \in \mathit{Context}$*:*

- $t$ *can be split as* $c[r]$ *using the evaluation context CFG of* RSEC*;*
- $\mathcal{R}^{\square}_{\text{RSEC}} \vdash \mathit{split}(t) \to c[r]$*;*
- $\mathcal{R}^{\square}_{\text{RSEC}} \vdash \mathit{plug}(c[r]) = t$*.*

The theorem above says that the process of splitting a term $t$ into a context and a redex in reduction semantics with evaluation contexts, which can be non-deterministic, reduces to reachability in the corresponding rewriting logic theory of a contextual representation pattern $c[r]$ of the original term marked for splitting, $\mathit{split}(t)$. Rewrite engines such as Maude provide a search command that does precisely that. We will shortly see how Maude's search command can find all splits of a term.

**rules:**
// for each reduction semantics rule $l(c_1[l_1], \ldots, c_n[l_n]) \to r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$
// add the following conditional semantic rewrite rule:

$$\circ\,\overline{l}(T_1, \ldots, T_n) \to \overline{r}(plug(\overline{c'_1}[\overline{r_1}]), \ldots, plug(\overline{c'_n}[\overline{r_{n'}}])) \text{ if } split(T_1) \to \overline{c_1}[\overline{l_1}] \wedge \ \ldots \ \wedge split(T_n) \to \overline{c_n}[\overline{l_n}]$$

Figure 3.33: First embedding of RSEC into rewriting logic (RSEC $\rightsquigarrow \mathcal{R}^{\boxed{1}}_{\text{RSEC}}$).

**Faithful Embedding of RSEC in Rewriting Logic**

In this section we discuss three faithful rewriting logic embeddings of reduction semantics with evaluation contexts. The first two assume that the embedded reduction semantics has no characteristic rule, in that all reductions take place at the top of the original term to reduce (e.g., a configuration in the case of our IMP language); this is not a limitation because, as already discussed, the characteristic rule can be regarded as syntactic sugar anyway, its role being to allow one to write reduction semantics definitions more compactly and elegantly. The first embedding is the simplest and easiest to prove correct, but it is the heaviest in notation and the resulting rewrite theories tend to be inefficient when executed because most of the left-hand-side terms of rules end up being identical, thus making the task of matching and selecting a rule to apply rather complex for rewrite engines. The second embedding results in rewrite rules whose left-hand-side terms are mostly distinct, thus taking advantage of current strengths of rewrite engines to index terms so that rules to be applied can be searched for quickly. Our third embedding is as close to the original reduction semantics in form and shape as one can hope it to be in a rewriting setting; in particular, it also defines a characteristic rule, which can be used to write a more compact semantics. The third embedding yields rewrite theories which are as efficient as those produced by the second embedding. The reason we did not define directly the third embedding is because we believe that the transition from the first to the second and then to the third is instructive.

Since reduction semantics with evaluation contexts is an inherently small-step semantical approach, we use the same mechanism to control the rewriting as for small-step SOS (Section 3.3) and MSOS (Section 3.6). This mechanism was discussed in detail in Section 3.3.3. It essentially consists of: (1) tagging each left-hand-side term appearing in a rule transition with a $\circ$, to capture the desired notion of a one-step reduction of that term; and (2) tagging with a $\star$ the terms to be multi-step (zero, one or more steps) reduced, where $\star$ can be easily defined with a conditional rule as the transitive and reflexive closure of $\circ$ (see Section 3.3.3).

Figure 3.33 shows our first embedding of reduction semantics with evaluation contexts into rewriting logic, which assumes that the characteristic rule, if any, has already been desugared. Each reduction semantics rule translates into one conditional rewrite rule. We allow the reduction rules to have in their left-hand-side and right-hand-side terms an arbitrary number of subterms that are in contextual representation. For example, if the left-hand side $l$ of a reduction rule has $n$ such subterms, say $c_1[l_1]$, ..., $c_n[l_n]$, then we write it $l(c_1[l_1], \ldots, c_n[l_n])$ (this is similar with our previous notation $\pi(N_1, \ldots, N_n, N)$ in the section above on embedding of evaluation contexts into rewriting logic, except that we now single out all the subterms in contextual representation instead of all the non-terminals). In particular, a rule $l \to r$ in which $l$ and $r$ contain no subterms in contextual representation (like the last rule in Figure 3.31) is translated exactly like in small-step SOS, that is,

into $\overline{l} \to \overline{r}$. Also, note that we allow evaluation contexts to have any pattern (since we overline them, like any other terms); we do not restrict them to only be context variables. Consider, for example, the six reduction rules discussed in the preamble of Section 3.7, which after the desugaring of the characteristic rule are as follows:

$$\begin{aligned}
c[i_1 \texttt{ <= } i_2] &\to c[i_1 \leq_{Int} i_2] \\
c[\texttt{ skip ; } s_2] &\to c[s_2] \\
c[\texttt{ if true then } s_1 \texttt{ else } s_2] &\to c[s_1] \\
c[\texttt{ if false then } s_1 \texttt{ else } s_2] &\to c[s_2] \\
\langle c, \sigma \rangle[x] &\to \langle c, \sigma \rangle[\sigma(x)] \quad \text{if } \sigma(x) \neq \bot \\
\langle c, \sigma \rangle[x \texttt{ := } i] &\to \langle c, \sigma[i/x] \rangle[\texttt{ skip }] \quad \text{if } \sigma(x) \neq \bot
\end{aligned}$$

Since all these rules have left-hand side-terms already in contextual representation, their corresponding $l$ in Figure 3.33 is just a non-terminal (*Configuration*), which means that $\overline{l}$ is just a variable (of sort *Configuration*). Therefore, the rewriting logic rules associated to these RSEC rules are:

$$\begin{aligned}
&\circ \mathit{Cfg} \to \mathit{plug}(C[I_1 \leq_{Int} I_2]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to C[I_1 \texttt{ <= } I_2] \\
&\circ \mathit{Cfg} \to \mathit{plug}(C[S_2]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to C[\texttt{ skip ; } S_2] \\
&\circ \mathit{Cfg} \to \mathit{plug}(C[S_1]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to C[\texttt{ if true then } S_1 \texttt{ else } S_2] \\
&\circ \mathit{Cfg} \to \mathit{plug}(C[S_2]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to C[\texttt{ if false then } S_1 \texttt{ else } S_2] \\
&\circ \mathit{Cfg} \to \mathit{plug}(\langle C, \sigma \rangle[\sigma(X)]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to \langle C, \sigma \rangle[X] \ \wedge \ \sigma(X) \neq \bot \\
&\circ \mathit{Cfg} \to \mathit{plug}(\langle C, \sigma[I/X] \rangle[\texttt{ skip }]) \textbf{ if } \mathit{split}(\mathit{Cfg}) \to \langle C, \sigma \rangle[X \texttt{ := } I] \ \wedge \ \sigma(X) \neq \bot
\end{aligned}$$

Recall from the preamble of Section 3.7 that the RSEC rules for variable lookup and assignment can also be given as follows, so that their left-hand-side terms are not in contextual representation:

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

In these cases, the string $l$ in Figure 3.33 has the form $\langle \mathit{Stmt}, \sigma \rangle$, which means that $\overline{l}$ is the term $\langle S, \sigma \rangle$, where $S$ is a variable of sort *Stmt*. Then their corresponding rewriting logic rules are:

$$\begin{aligned}
&\circ \langle S, \sigma \rangle \to \langle \mathit{plug}(C[\sigma(X)]), \sigma \rangle \textbf{ if } \mathit{split}(S) \to C[X] \ \wedge \ \sigma(X) \neq \bot \\
&\circ \langle S, \sigma \rangle \to \langle \mathit{plug}(C[\texttt{ skip }]), \sigma[I/X] \rangle \textbf{ if } \mathit{split}(S) \to C[X \texttt{ := } I] \ \wedge \ \sigma(X) \neq \bot
\end{aligned}$$

Once the characteristic rule is desugared as explained in the preamble of Section 3.7, an RSEC rule operates as follows: (1) attempt to match the left-hand-side pattern of the rule at the top of the term to reduce, making sure that each of the subterms corresponding to subpatterns in contextual representation form can indeed be split as indicated; and (2) if the matching step above succeeds, then reduce the original term to the right-hand-side pattern instantiated accordingly, plugging all the subterms appearing in contextual representations in the right-hand side. Note that the conditional rewrite rule associated to an RSEC rule as indicated in Figure 3.33 achieves precisely the desired steps above: the $\circ$ in the left-hand-side term guarantees that the rewrite step takes place at the top of the original term, the condition exhaustively searches for the desired splits of the subterms in question into contextual representations, and the right-hand side plugs back all the contextual representations into terms over the original syntax. The only difference between the original RSEC rule and its corresponding rewriting logic conditional rule is that the rewriting logic rule makes explicit the splits and plugs that are implicit in the RSEC rule.

**Theorem 7.** ***(First faithful embedding of reduction semantics into rewriting logic)*** *Let* RSEC *be any reduction semantics with evaluation contexts definition and let* $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ *be the rewriting logic theory associated to* RSEC *using the embedding procedures in Figures 3.32 and 3.33. Then*

1. ***(step-for-step correspondence)*** $\text{RSEC} \vdash t \to t'$ *using a reduction semantics with evaluation contexts rule iff* $\mathcal{R}^{\boxed{1}}_{\text{RSEC}} \vdash \circ\, \overline{t} \to^1 \overline{t'}$ *using the corresponding conditional rewrite rule obtained like in Figure 3.33; moreover, the reduction rule and the corresponding rewrite rule apply similarly (same contexts, same substitution; all modulo the correspondence in Theorem 6);*

2. ***(computational correspondence)*** $\text{RSEC} \vdash t \to^\star t'$ *iff* $\mathcal{R}^{\boxed{1}}_{\text{RSEC}} \vdash \star\, \overline{t} \to \star\, \overline{t'}$.

The first item in Theorem 7 says that the resulting rewriting logic theory captures faithfully the small-step reduction relation of the original reduction semantics with evaluation contexts definition. The faithfulness of this embedding (i.e., there is precisely one top-level application of a rewrite rule that corresponds to an application of a reduction semantics rule), comes from the fact that the consistent use of the $\circ$ tag inhibits any other application of any other rule on the tagged term. Therefore, like in small-step SOS and MSOS, a small-step in a reduction semantics definition also reduces to reachability analysis in the corresponding rewrite theory; one can also use the search capability of a system like Maude to find all the next terms that a given term evaluates to (Maude provides the capability to search for the first $n$ terms that match a given pattern using up to $m$ rule applications, where $n$ and $m$ are user-provided parameters).

The step-for-step correspondence above is stronger (and better) than the strong bisimilarity of the two definitions; for example, if a reduction semantics rule in RSEC can be applied in two different ways on a term to reduce, then its corresponding rewrite rule in $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ can also be applied in two different ways on the tagged term. The second item in Theorem 7 says that the resulting rewrite theory can be used to perform any computation possible in the original RSEC, and vice versa (the step-for-step correspondence is guaranteed in combination with the first item). Therefore, there is absolutely no difference between computations using RSEC and computations using $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$, except for irrelevant syntactic conventions/notations. This strong correspondence between reductions in RSEC and rewrites in $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ tells that $\mathcal{R}^{\boxed{1}}_{\text{RSEC}}$ *is precisely* RSEC*, not an encoding of it.* In other words, RSEC can be faithfully regarded as a methodological fragment of rewriting logic, same like big-step SOS, small-step SOS, and MSOS.

The discussion above implies that, from a theoretical perspective, the rewriting logic embedding of reduction semantics in Figure 3.33 is as good as one can hope. However, its simplicity comes at a price in performance, which unfortunately tends to be at its worst precisely in the most common cases. Consider, for example, the six rewrite rules used before Theorem 7 to exemplify the embedding in Figure 3.33 (consider the variant for lookup and assignment rules where the contextual representation in the left-hand side appears at the top—first variant). They all have the form:

$$\circ\, \mathit{Cfg} \to \ldots \;\; \textbf{if} \;\; \mathit{split}(\mathit{Cfg}) \to \ldots$$

In fact, as seen in Figure 3.38, all the rewrite rules in the rewriting logic theory corresponding to the RSEC of IMP have the same form. The reason the left-hand-side terms of these rewrite rules are the same and lack any structure is because the contextual representations in the left-hand-side terms of the RSEC rules appear at the top, with no structure above them, which is the most common type of RSEC rule encountered.

To apply a conditional rewrite rule, a rewrite engine first matches the left-hand side and then performs the (exhaustive) search in the condition. In other words, the structure of the left-hand side acts as a cheap guard for the expensive search. Unfortunately, since the left-hand side of the conditional rewrite rules above has no structure, it will always match. That means that the searches in the conditions of all the rewrite rules will be, in the worst case, executed one after another until a split is eventually found (if any). If one thinks in terms of implementing RSEC in general, then this is what a naive implementation would do. If one thinks in terms of executing term rewrite systems, then this fails to take advantage of some important performance-increasing advances in term rewriting, such as *indexing* [82, 83, 3]. In short, indexing techniques use the structure of the left-hand sides to augment the term structure with information about which rule can potentially be applied at which places. This information is dynamically updated, as the term is rewritten. If the rules' left-hand sides do not significantly overlap, it is generally assumed that it takes constant time to find a matching rewrite rule. This is similar in spirit to hashing, where the access time into a hash table is generally assumed to take constant time when there are no or few key collisions. Thinking intuitively in terms of hashing, from an indexing perspective a rewrite system with rules having the same left-hand sides is as bad as a hash table in which all accesses are collisions.

Ideally, in an efficient implementation of RSEC one would like to adapt/modify indexing techniques, which currently work for context-insensitive term rewriting, or to invent new techniques serving the same purpose. This seems highly non-trivial and tedious, though. An alternative is to device embedding transformations of RSEC into rewriting logic that take better or full advantage of existing, context-insensitive indexing. Without context-sensitive indexing or other bookkeeping mechanisms hardwired in the reduction engine, due to the inherent non-determinism in parsing/splitting syntax into contextual representations, in the worst case one needs to search the entire term to find a legal position where a reduction can take place. While there does not seem that we can do much to avoid such an exhaustive search in the worst case, note that our first embedding in Figure 3.33 initiates such a search in the condition of every rewrite rule: since in practice many/most of the rewrite rules generated by the procedure in Figure 3.33 end up having the same left-hand side, the expensive search for appropriate splittings is potentially invoked many times. What we'd like to achieve is: (1) activate the expensive search for splitting only once; and (2) for each found split, quickly test which rule applies and apply it. Such a quick test as desired in (2) can be achieved for free on existing rewrite systems that use indexing, such as Maude, if one slightly modifies the embedding translation of RSEC into rewriting logic as shown in Figure 3.34.

The main idea is to keep the structure of the left-hand side of the RSEC rules in the left-hand side of the corresponding rewrite rules. This structure is crucial for indexing. To allow it, one needs to do the necessary splitting as a separate step. The first type of rewrite rules in Figure 3.34, one per term appearing as a left-hand side in any of the conditional rules generated following the first embedding in Figure 3.33, enables the splitting process on the corresponding contextual representations in the left-hand side of the original RSEC rule. We only define such rules for left-hand-side terms having at least one subterm in contextual representation, because if the left-hand side $l$ has no such terms then the rule would be $\circ\bar{l} \to T$ **if** $\circ\bar{l} \to T$, which is useless and does not terminate.

The second type of rules in Figure 3.34, one per RSEC rule, have almost the same left-hand sides as the original RSEC rules; the only difference is the algebraic notation (as reflected by the overlining). Their right-hand sides plug the context representations, so that they always yield terms which are well-formed over the original syntax (possibly extended with auxiliary syntax for semantics components—configurations, states, etc.). Consider, for example, the six RSEC rules

**rules:**

// for each term $l$ that appears as left-hand side of a reduction rule
// $l(c_1[l_1], \ldots, c_n[l_n]) \to \ldots$ with $n > 0$, add the following
// conditional rewrite rule (there could be one $l$ for many reduction rules):

$$\circ\, \overline{l}(T_1, \ldots, T_n) \to T \;\; \textbf{if} \;\; \circ\, \overline{l}(\mathit{split}(T_1), \ldots, \mathit{split}(T_n)) \to T$$

// for each reduction semantics rule $l(c_1[l_1], \ldots, c_n[l_n]) \to r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$
// add the following (unconditional) semantic rewrite rule:

$$\circ\, \overline{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}]) \to \overline{r}(\mathit{plug}(\overline{c'_1}[\overline{r_1}]), \ldots, \mathit{plug}(\overline{c'_n}[\overline{r_{n'}}]))$$

Figure 3.34: Second embedding of RSEC into rewriting logic (RSEC $\rightsquigarrow \mathcal{R}^{\boxed{2}}_{\mathrm{RSEC}}$).

discussed in the preamble of Section 3.7, whose translation into rewrite rules following our first embedding in Figure 3.33 was discussed right above Theorem 7. Let us first consider the variant for lookup and assignment rules where the contextual representation in the left-hand side appears at the top. Since in all these rules the contextual representation appears at the top of their left-hand side, which in terms of the first embedding in Figure 3.33 means that their corresponding rewrite rules (in the first embedding) had the form $\circ\, \mathit{Cfg} \to \ldots$ **if** $\mathit{split}(\mathit{Cfg}) \to \ldots$, we only need to add one rule of the first type in Figure 3.34 for them, namely ($l$ is the identity pattern, i.e., $\overline{l}$ is a variable):

$$\circ\, \mathit{Cfg} \to \mathit{Cfg}' \;\; \textbf{if} \;\; \circ\, \mathit{split}(\mathit{Cfg}) \to \mathit{Cfg}'$$

With this, the six rewrite rules of the second type in Figure 3.34 corresponding to the six RSEC rules under discussion are the following:

$$\begin{aligned}
\circ\, C[I_1 \texttt{ <= } I_2] &\to \mathit{plug}(C[I_1 \leq_{\mathit{Int}} I_2]) \\
\circ\, C[\texttt{ skip ; } S_2] &\to \mathit{plug}(C[S_2]) \\
\circ\, C[\texttt{ if true then } S_1 \texttt{ else } S_2] &\to \mathit{plug}(C[S_1]) \\
\circ\, C[\texttt{ if false then } S_1 \texttt{ else } S_2] &\to \mathit{plug}(C[S_2]) \\
\circ\, \langle C, \sigma\rangle[X] &\to \mathit{plug}(\langle C, \sigma\rangle[\sigma(X)]) \;\; \textbf{if} \;\; \sigma(X) \neq \bot \\
\circ\, \langle C, \sigma\rangle[X \texttt{ := } I] &\to \mathit{plug}(\langle C, \sigma[I/X]\rangle[\texttt{ skip }]) \;\; \textbf{if} \;\; \sigma(X) \neq \bot
\end{aligned}$$

If one prefers the second variant for the reduction rules of lookup and assignment, namely

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

then, since the left-hand side of these rules is a pattern of the form $\langle \mathit{Stmt}, \sigma\rangle$ which in algebraic form (over-lined) becomes a term of the form $\langle S, \sigma\rangle$, we need to add one more rewrite rule of the first type in Figure 3.34, namely

$$\circ\, \langle S, \sigma\rangle \to \mathit{Cfg}' \;\; \textbf{if} \;\; \circ\, \langle \mathit{split}(S), \sigma\rangle \to \mathit{Cfg}',$$

and to replace the rewrite rules for lookup and assignment above with the following two rules:

$$\begin{aligned}
\circ\, \langle C[X], \sigma\rangle &\to \langle \mathit{plug}(C[\sigma(X)]), \sigma\rangle \;\; \textbf{if} \;\; \sigma(X) \neq \bot \\
\circ\, \langle C[X \texttt{ := } I], \sigma\rangle &\to \langle \mathit{plug}(C[\texttt{ skip }]), \sigma[I/X]\rangle \;\; \textbf{if} \;\; \sigma(X) \neq \bot
\end{aligned}$$

**Theorem 8.** ***(Second faithful embedding of reduction semantics in rewriting logic)*** *Let* RSEC *be any reduction semantics with evaluation contexts definition and let* $\mathcal{R}^{\boxed{2}}_{\text{RSEC}}$ *be the rewriting logic theory associated to* RSEC *using the embedding procedures in Figures 3.32 and 3.34. Then*

1. ***(step-for-step correspondence)*** $\text{RSEC} \vdash t \to t'$ *using a reduction semantics with evaluation contexts rule iff* $\mathcal{R}^{\boxed{2}}_{\text{RSEC}} \vdash \circ\,\overline{t} \to^1 \overline{t'}$ *using the corresponding rewrite rules obtained like in Figure 3.34 (first a conditional rule of the first type whose left-hand side matches* $t$*, then a rule of the second type which solves, in one rewrite step, the condition of the first rule); moreover, the reduction rule and the corresponding rewrite rules apply similarly (same contexts, same substitution; all modulo the correspondence in Theorem 6);*

2. ***(computational correspondence)*** $\text{RSEC} \vdash t \to^\star t'$ *iff* $\mathcal{R}^{\boxed{2}}_{\text{RSEC}} \vdash \star\,\overline{t} \to \star\,\overline{t'}$.

Theorem 8 tells us that we can use our second rewriting logic embedding transformation in Figure 3.34 to seamlessly execute RSEC definitions on context-insensitive rewrite engines, such as Maude. This was also the case for our first embedding (Figure 3.33 and its corresponding Theorem 7). However, as explained above, in our second embedding the left-hand-side terms of the rewrite rules corresponding to the actual reduction semantics rules (the second type of rule in Figure 3.34) preserve the structure of the left-hand-side terms of the original corresponding reduction rules. This important fact has two benefits. On the one hand, the underlying rewrite engines can use that structure to enhance the efficiency of rewriting by means of indexing, as already discussed above. On the other hand, the resulting rewrite rules resemble the original reduction rules, so the language designer who wants to use our embedding feels more comfortable. Indeed, since the algebraic representation of terms (the overline) should not change the way they are perceived by a user, the only difference between the left-hand side of the original reduction rule and the left-hand side of the resulting rewrite rule is the $\circ$ symbol: $l(c_1[l_1], \ldots, c_n[l_n])$ versus $\circ\,\overline{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}])$, e.g., $\langle c[x := i], \sigma\rangle$ versus $\circ\,\langle C[X := I], \sigma\rangle$, where $c$, $\sigma$, $x$, $i$ are reduction rule parameters while $C$, $\sigma$, $X$, $I$ are corresponding variables of appropriate sorts.

Even though the representational distance between the left-hand-side terms in the original reduction rules and the left-hand-side terms in the resulting rewrite rules is minimal (one cannot eliminate the $\circ$, as extensively discussed in Section 3.3.3), unfortunately, the same does not hold true for the right-hand-side terms. Indeed, a right-hand side $r(c'_1[r_1], \ldots, c'_n[r_{n'}])$ of a reduction rule becomes the right-hand side $\overline{r}(\mathit{plug}(\overline{c'_1}[\overline{r_1}]), \ldots, \mathit{plug}(\overline{c'_n}[\overline{r_{n'}}]))$ of its corresponding rewrite rule, e.g., $\langle c[\,\texttt{skip}\,], \sigma\rangle$ becomes $\langle \mathit{plug}(C[\,\texttt{skip}\,]), \sigma\rangle$.

Figure 3.35 shows our third and final embedding of RSEC in rewriting logic, which has the advantage that it completely isolates the uses of *split*/*plug* from the semantic rewrite rules. Indeed, the rewrite rule associated to a reduction rule has the same left-hand side as in the second embedding, but now the right-hand side is actually the algebraic variant of the right-hand side of the original reduction rule. This is possible because of two simple adjustments of the second embedding:

1. To avoid having to explicitly use the *plug* operation in the semantic rewrite rules, we replace the first type of conditional rewrite rules in the second embedding, namely

$$\circ\,\overline{l}(T_1, \ldots, T_n) \to T \;\; \textbf{if} \;\; \circ\,\overline{l}(\mathit{split}(T_1), \ldots, \mathit{split}(T_n)) \to T,$$

   with slightly modified conditional rewrite rules of the form

$$\circ\,\overline{l}(T_1, \ldots, T_n) \to T \;\; \textbf{if} \;\; \mathit{plug}(\circ\,\overline{l}(\mathit{split}(T_1), \ldots, \mathit{split}(T_n))) \to T.$$

**rules:**

// for each term $l$ that appears as the left-hand side of a reduction rule
// $l(c_1[l_1], \ldots, c_n[l_n]) \to \ldots$, add the following conditional
// rewrite rule (there could be one $l$ for many reduction rules):

$$\circ\,\overline{l}(T_1, \ldots, T_n) \to T \;\; \textbf{if} \;\; plug(\circ\,\overline{l}(split(T_1), \ldots, split(T_n))) \to T$$

// for each non-identity term $r$ appearing as right-hand side in a reduction rule
// $\ldots \to r(c_1[r_1], \ldots, c_n[r_n])$, add the following equation
// (there could be one $r$ for many reduction rules):

$$plug(\overline{r}(Syn_1, \ldots, Syn_n)) = \overline{r}(plug(Syn_1), \ldots, plug(Syn_n))$$

// for each reduction semantics rule $l(c_1[l_1], \ldots, c_n[l_n]) \to r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$
// add the following semantic rewrite rule:

$$\circ\,\overline{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}]) \to \overline{r}(\overline{c'_1}[\overline{r_1}], \ldots, \overline{c'_n}[\overline{r_{n'}}])$$

Figure 3.35: Third embedding of RSEC in rewriting logic (RSEC $\rightsquigarrow \mathcal{R}^{\boxed{3}}_{\mathrm{RSEC}}$).

Therefore, the left-hand-side term of the condition is wrapped with the *plug* operation. Since rewriting is context-insensitive, the *plug* wrapper does not affect the rewrites that happen underneath in the $\circ\,\overline{l}(\ldots)$ term. Like in the second embedding, the only way for $\circ$ to disappear from the condition left-hand side is for a semantic rule to apply. When that happens, the left-hand side of the condition is rewritten to a term of the form $plug(t)$, where $t$ matches the right-hand side of some reduction semantics rule, which may potentially contain some subterms in contextual representation.

2. To automatically plug all the subterms in contextual representation that appear in $t$ after the left-hand-side term of the condition in the rule above rewrites to $plug(t)$, we add equations of the form

$$plug(\overline{r}(Syn_1, \ldots, Syn_n)) = \overline{r}(plug(Syn_1), \ldots, plug(Syn_n)),$$

one for each non-identity pattern $r$ appearing as a right-hand side of an RSEC rule; if $r$ is an identity pattern then the equation becomes $plug(Syn) = plug(Syn)$, so we omit it.

Let us exemplify our third rewriting logic embedding transformation of reduction semantics with evaluation contexts using the same six reduction rules used so far in this section, but, to make it more interesting, considering the second variant of reduction rules for variable lookup and assignment. We have two left-hand-side patterns in these reduction rules, namely *Configuration* and $\langle Stmt, \sigma \rangle$, so we have the following two rules of the first type in Figure 3.35:

$$\begin{array}{c} \circ\, Cfg \to Cfg' \;\; \textbf{if} \;\; plug(\circ\, split(Cfg)) \to Cfg' \\ \circ\, \langle S, \sigma \rangle \to Cfg' \;\; \textbf{if} \;\; plug(\circ\, \langle split(S), \sigma \rangle) \to Cfg' \end{array}$$

We also have two right-hand-side patterns in these reduction rules, the same two as above, but the

first one is an identity pattern so we only add one equation of the second type in Figure 3.35:

$$plug(\langle C[Syn], \sigma\rangle) = \langle plug(C[Syn]), \sigma\rangle$$

We can now give the six rewrite rules corresponding to the six reduction rules in discussion:

$$\begin{aligned}
\circ\, C[I_1 \texttt{ <= } I_2] &\to C[I_1 \leq_{Int} I_2] \\
\circ\, C[\texttt{ skip ; } S_2] &\to C[S_2] \\
\circ\, C[\texttt{ if true then } S_1 \texttt{ else } S_2] &\to C[S_1] \\
\circ\, C[\texttt{ if false then } S_1 \texttt{ else } S_2] &\to C[S_2] \\
\circ\, \langle C[X], \sigma\rangle &\to \langle C[\sigma(X)], \sigma\rangle \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ\, \langle C[X \texttt{ := } I], \sigma\rangle &\to \langle C[\texttt{ skip }], \sigma[I/X]\rangle \ \textbf{if}\ \sigma(X) \neq \bot
\end{aligned}$$

The six rewrite rules above are as close to the original reduction semantics rules as one can hope them to be in a rewriting setting. Note that, for simplicity, we preferred to desugar the characteristic rule of reduction semantics with evaluation contexts in all our examples in this subsection. At this moment we have all the infrastructure needed to also include a rewrite equivalent of it:

$$\circ\, C[Syn] \to C[Syn'] \ \textbf{if}\ C \neq \Box \wedge \circ\, Syn \to Syn'$$

Note that we first check whether the context is proper in the condition of the characteristic rewrite rule above, and then we initiate a (small-step) reduction of the redex (by tagging it with the symbol $\circ$). The condition is well-defined in rewriting logic because, as explained in Figure 3.32, we subsorted all the syntactic sorts together with the configuration under the top sort *Syntax*, so all these sorts belong to the same kind (see Section 2.7), which means that the operation $\circ$ can apply to any of them, including to *Syntax*, despite the fact that it was declared to take a *Configuration* to an *ExtendedConfiguration* (like in Section 3.3.3). With this characteristic rewrite rule, we can now restate the six rewrite rules corresponding to the six reduction rules above as follows:

$$\begin{aligned}
\circ\, I_1 \texttt{ <= } I_2 &\to I_1 \leq_{Int} I_2 \\
\circ\, \texttt{ skip ; } S_2 &\to S_2 \\
\circ\, \texttt{ if true then } S_1 \texttt{ else } S_2 &\to S_1 \\
\circ\, \texttt{ if false then } S_1 \texttt{ else } S_2 &\to S_2 \\
\circ\, \langle C[X], \sigma\rangle &\to \langle C[\sigma(X)], \sigma\rangle \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ\, \langle C[X \texttt{ := } I], \sigma\rangle &\to \langle C[\texttt{ skip }], \sigma[I/X]\rangle \ \textbf{if}\ \sigma(X) \neq \bot
\end{aligned}$$

Note, again, that $\circ$ is applied on arguments of various sorts in the same kind with *Configuration*.

The need for $\circ$ in the left-hand-side terms of rules like above is now even more imperative than before. In addition to all the reasons discussed so far, there are additional reasons now for which the dropping of $\circ$ would depart us from the intended faithful capturing of reduction semantics in rewriting logic. Indeed, if we drop $\circ$ then there is nothing to stop the applications of rewrite rules at any places in the term to rewrite, potentially including places which are not allowed to be evaluated yet, such as, for example, in the branches of a conditional. Moreover, such applications of rules could happen concurrently, which is strictly disallowed by reduction semantics with or without evaluation contexts. The role of $\circ$ is precisely to inhibit the otherwise unrestricted potential to apply rewrite rules everywhere and concurrently: rules are now applied sequentially and only at the top of the original term, exactly like in reduction semantics.

**sorts:**
$\mathit{Configuration}$, $\mathit{ExtendedConfiguration}$
**subsort:**
$\mathit{Configuration} < \mathit{ExtendedConfiguration}$
**operations:**
$\langle \_, \_ \rangle$ : $\mathit{Stmt} \times \mathit{State} \to \mathit{Configuration}$
$\langle \_ \rangle$ : $\mathit{Pgm} \to \mathit{Configuration}$
$\circ \_$ : $\mathit{Configuration} \to \mathit{ExtendedConfiguration}$ // reduce one step
$\star \_$ : $\mathit{Configuration} \to \mathit{ExtendedConfiguration}$ // reduce all steps
**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 3.36: Configurations and infrastructure for the rewriting logic embedding of RSEC(IMP).

**Theorem 9.** ***(Third faithful embedding of reduction semantics into rewriting logic)*** *Let* RSEC *be any reduction semantics with evaluation contexts definition (with or without a characteristic reduction rule) and let* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC}}$ *be the rewriting logic theory associated to* RSEC *using the embedding procedures in Figures 3.32 and 3.35 (plus the characteristic rewrite rule above in case* RSEC *comes with a characteristic reduction rule). Then*

1. ***(step-for-step correspondence)*** $\mathrm{RSEC} \vdash t \to t'$ *using a reduction semantics with evaluation contexts rule iff* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC}} \vdash \circ\, \overline{t} \to^1 \overline{t'}$*;*

2. ***(computational correspondence)*** $\mathrm{RSEC} \vdash t \to^\star t'$ *iff* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC}} \vdash \star\, \overline{t} \to \star\, \overline{t'}$*.*

We can therefore safely conclude that RSEC has been captured as a methodological fragment of rewriting logic. The faithful embeddings of reduction semantics into rewriting logic above can be used in at least two different ways. On the one hand, they can be used as compilation steps transforming a context-sensitive reduction system into an equivalent context-insensitive rewrite system, which can be further executed/compiled/analyzed using conventional rewrite techniques and existing rewrite engines. On the other hand, the embeddings above are so simple, that one can simply use them manually and thus "think reduction semantics" in rewriting logic.

### Reduction Semantics with Evaluation Contexts of IMP in Rewriting Logic

We here discuss the complete reduction semantics with evaluation contexts definition of IMP in rewriting logic, obtained by applying the faithful embedding techniques discussed above to the reduction semantics definition of IMP in Figure 3.31 in Section 3.7.1. We start by defining the needed configurations, then we give all the rewrite rules and equations embedding the evaluation contexts and their splitting/plugging mechanism in rewriting logic, and then we finally give three rewrite theories corresponding to the three embeddings discussed above, each including the (same) configurations definition and embedding of evaluation contexts.

Figure 3.36 gives an algebraic definition of IMP configurations as needed for reduction semantics with evaluation contexts, together with the additional infrastructure needed to represent the one-step and multi-step transition relations. Everything defined in Figure 3.36 has already been discussed

in the context of small-step SOS (see Figures 3.13 and 3.17 in Section 3.3.3). Note, however, that we only defined a subset of the configurations needed for small-step SOS, more precisely only the top-level configurations (ones holding a program and ones holding a statement and a state). The intermediate configurations holding expressions and a state in small-step SOS are not needed here because reduction semantics with evaluation contexts does not need to explicitly decompose bigger reduction tasks into smaller ones until a redex is eventually found, like small-step SOS does; instead, the redex is found atomically by splitting the top level configuration into a context and the redex.

Figure 3.37 shows the rewriting logic theory $\mathcal{R}^{\Box}_{\text{RSEC(IMP)}}$ associated to the evaluation contexts of IMP in RSEC(IMP) (Figure 3.30) following the procedure described in Section 3.7.3 and summarized in Figure 3.32. Recall that all language syntactic categories and configurations are sunk into a top sort *Syntax*, and that one rule for splitting and one equation for plugging are generated for each context production. In general, the embedding of evaluation contexts tends to be the largest and the most boring portion of the rewriting logic embedding of a reduction semantics language definition. However, fortunately, this can be generated fully automatically. An implementation of the rewriting logic embedding techniques discussed in this section may even completely hide this portion from the user. We show it in Figure 3.37 only for the sake of completeness.

Figure 3.38 shows the rewriting logic theory $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ corresponding to the rules in the reduction semantics with evaluation contexts of IMP in Section 3.7.1, following our first embedding transformation depicted in Figure 3.33. 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., $\Sigma$ instead of $\sigma$ for variables of sort *State*). These rules are added, of course, to those corresponding to evaluation contexts in Figure 3.37 (which are common to all three embeddings). Note that there is precisely one conditional rewrite rule in Figure 3.38 corresponding to each reduction semantics rule of IMP in Figure 3.31. Also, note that if a rule does not make use of evaluation contexts, then its corresponding rewrite rule is identical to the rewrite rule corresponding to the small-step SOS embedding discussed in Section 3.3.3. For example, the last reduction rule in Figure 3.31 results in the last rewrite rule in Figure 3.38, which is identical to the last rewrite rule corresponding to the small-step SOS of IMP in Figure 3.18. The rules that make use of evaluation contexts perform explicit splitting (in the left-hand side of the condition) and plugging (in the right-hand side of the conclusion) operations. As already discussed but worth reemphasizing, the main drawbacks of this type of rewriting logic embedding are: (1) the expensive, non-deterministic search involving splitting of the original term is performed for any rule, and (2) it does not take advantage of one of the major optimizations of rewrite engines, indexing, which allows for quick detection of matching rules based on the structure of their left-hand-side terms.

Figure 3.39 shows the rewriting logic theory $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ that follows our second embedding transformation depicted in Figure 3.34. These rules are also added to those corresponding to evaluation contexts in Figure 3.37. Note that now there is precisely one *unconditional* rewrite rule corresponding to each reduction semantics rule of IMP in Figure 3.31 and that, unlike in the first embedding in Figure 3.38, the left-hand side of each rule preserves the exact structure of the left-hand side of the original reduction rule (after desugaring of the characteristic rule), so this embedding takes advantage of indexing optimizations in rewrite engines. Like in the first embedding, if a reduction rule does not make use of evaluation contexts, then its corresponding rewrite rule is identical to the rewrite rule corresponding to the small-step SOS embedding discussed in Section 3.3.3 (e.g., the last rule). Unlike in the first embedding, we also need to add a generic conditional rule, the first one in Figure 3.39, which initiates the splitting. We need only one rule of this type because

**sorts:**

$Syntax,\ Context$

**subsorts:**

$AExp, BExp, Stmt, Configuration < Syntax$

**operations:**

$\square : \to Context$ $\qquad$ $\_[\_] : Context \times Syntax \to Syntax$

$split : Syntax \to Syntax$ $\qquad$ $plug : Syntax \to Syntax$

$\langle \_, \_ \rangle : Context \times State \to Context$

$\_ + \_ : Context \times AExp \to Context$ $\qquad$ $\_ + \_ : AExp \times Context \to Context$

$\_ / \_ : Context \times AExp \to Context$ $\qquad$ $\_ / \_ : AExp \times Context \to Context$

$\_ <= \_ : Context \times AExp \to Context$ $\qquad$ $\_ <= \_ : Int \times Context \to Context$

$\_ \texttt{and} \_ : Context \times BExp \to Context$

$\texttt{not} \_ : Context \to Context$

$\_ := \_ : Id \times Context \to Context$

$\_ ; \_ : Context \times Stmt \to Context$

$\texttt{if} \_ \texttt{then} \_ \texttt{else} \_ : Context \times Stmt \times Stmt \to Context$

**rules and equations:**

$split(Syn) \to \square[Syn]$ $\qquad$ $plug(\square[Syn]) = Syn$

$split(\langle S, \sigma \rangle) \to \langle C, \sigma \rangle[Syn]$ **if** $split(S) \to C[Syn]$

$plug(\langle C, \sigma \rangle[Syn]) = \langle plug(C[Syn]), \sigma \rangle$

$split(A_1 + A_2) \to (C + A_2)[Syn]$ **if** $split(A_1) \to C[Syn]$

$plug((C + A_2)[Syn]) = plug(C[Syn]) + A_2$

$split(A_1 + A_2) \to (A_1 + C)[Syn]$ **if** $split(A_2) \to C[Syn]$

$plug((A_1 + C)[Syn]) = A_1 + plug(C[Syn])$

$split(A_1 / A_2) \to (C / A_2)[Syn]$ **if** $split(A_1) \to C[Syn]$

$plug((C / A_2)[Syn]) = plug(C[Syn]) / A_2$

$split(A_1 / A_2) \to (A_1 / C)[Syn]$ **if** $split(A_2) \to C[Syn]$

$plug((A_1 / C)[Syn]) = A_1 / plug(C[Syn])$

$split(A_1 <= A_2) \to (C <= A_2)[Syn]$ **if** $split(A_1) \to C[Syn]$

$plug((C <= A_2)[Syn]) = plug(C[Syn]) <= A_2$

$split(I_1 <= A_2) \to (I_1 <= C)[Syn]$ **if** $split(A_2) \to C[Syn]$

$plug((I_1 <= C)[Syn]) = I_1 <= plug(C[Syn])$

$split(\texttt{not}\ B) \to (\texttt{not}\ C)[Syn]$ **if** $split(B) \to C[Syn]$

$plug((\texttt{not}\ C)[Syn]) = \texttt{not}\ plug(C[Syn])$

$split(B_1\ \texttt{and}\ B_2) \to (C\ \texttt{and}\ B_2)[Syn]$ **if** $split(B_1) \to C[Syn]$

$plug((C\ \texttt{and}\ B_2)[Syn]) = plug(C[Syn])\ \texttt{and}\ B_2$

$split(X := A) \to (X := C)[Syn]$ **if** $split(A) \to C[Syn]$

$plug((X := C)[Syn]) = X := plug(C[Syn])$

$split(S_1\ ;\ S_2) \to (C\ ;\ S_2)[Syn]$ **if** $split(S_1) \to C[Syn]$

$plug((C\ ;\ S_2)[Syn]) = plug(C[Syn])\ ;\ S_2$

$split(\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \to (\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2)[Syn]$ **if** $split(B) \to C[Syn]$

$plug((\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2)[Syn]) = \texttt{if}\ plug(C[Syn])\ \texttt{then}\ S_1\ \texttt{else}\ S_2$

Figure 3.37: $\mathcal{R}^{\square}_{\text{RSEC(IMP)}}$: Rewriting logic embedding of IMP evaluation contexts. The implicit split/plug reduction semantics mechanism is replaced by explicit rewriting logic sentences.

$\circ\ Cfg \to plug(\langle C, \sigma\rangle[\sigma(X)])$ **if** $split(Cfg) \to \langle C, \sigma\rangle[X] \ \wedge\ \sigma(X) \neq \bot$
$\circ\ Cfg \to plug(C[I_1 +_{Int} I_2])$ **if** $split(Cfg) \to C[I_1 + I_2]$
$\circ\ Cfg \to plug(C[I_1 /_{Int} I_2])$ **if** $split(Cfg) \to C[I_1 \,/\, I_2] \ \wedge\ I_2 \neq 0$
$\circ\ Cfg \to plug(C[I_1 \leq_{Int} I_2])$ **if** $split(Cfg) \to C[I_1 \texttt{<=} I_2]$
$\circ\ Cfg \to plug(C[\,\texttt{false}\,])$ **if** $split(Cfg) \to C[\,\texttt{not true}\,]$
$\circ\ Cfg \to plug(C[\,\texttt{true}\,])$ **if** $split(Cfg) \to C[\,\texttt{not false}\,]$
$\circ\ Cfg \to plug(C[B_2])$ **if** $split(Cfg) \to C[\,\texttt{true and}\, B_2]$
$\circ\ Cfg \to plug(C[\,\texttt{false}\,])$ **if** $split(Cfg) \to C[\,\texttt{false and}\, B_2]$
$\circ\ Cfg \to plug(\langle C, \sigma[I/X]\rangle[\,\texttt{skip}\,])$ **if** $split(Cfg) \to \langle C, \sigma\rangle[X := I] \ \wedge\ \sigma(X) \neq \bot$
$\circ\ Cfg \to plug(C[S_2])$ **if** $split(Cfg) \to C[\,\texttt{skip ;}\, S_2]$
$\circ\ Cfg \to plug(C[S_1])$ **if** $split(Cfg) \to C[\,\texttt{if true then}\, S_1\, \texttt{else}\, S_2]$
$\circ\ Cfg \to plug(C[S_2])$ **if** $split(Cfg) \to C[\,\texttt{if false then}\, S_1\, \texttt{else}\, S_2]$
$\circ\ Cfg \to plug(C[\,\texttt{if}\, B\, \texttt{then}\, (S\ \texttt{;}\ \texttt{while}\, B\, \texttt{do}\, S)\, \texttt{else skip}\,])$ **if** $split(Cfg) \to C[\,\texttt{while}\, B\, \texttt{do}\, S]$
$\circ\ \langle\, \texttt{var}\, Xl\ \texttt{;}\ S\rangle \to \langle S, (Xl \mapsto 0)\rangle$

Figure 3.38: $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ — rewriting logic theory corresponding to the first embedding of the reduction semantics with evaluation contexts of IMP.

$$\circ\ Cfg \to Cfg' \ \textbf{if}\ \circ\ split(Cfg) \to Cfg'$$

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

Figure 3.39: $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ — rewriting logic theory corresponding to the second embedding of the reduction semantics with evaluation contexts of IMP.

$$
\begin{aligned}
\circ\, \mathit{Cfg} &\to \mathit{Cfg}' \ \textbf{if}\ \circ\, \mathit{plug}(\mathit{split}(\mathit{Cfg})) \to \mathit{Cfg}' \\
\circ\, C[\mathit{Syn}] &\to C[\mathit{Syn}'] \ \textbf{if}\ C \neq \Box \wedge \circ\, \mathit{Syn} \to \mathit{Syn}' \\
\\
\circ\, \langle C, \sigma \rangle[X] &\to \langle C, \sigma \rangle[\sigma(X)] \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ\, I_1 \,\texttt{+}\, I_2 &\to I_1 +_{\mathit{Int}} I_2 \\
\circ\, I_1 \,\texttt{/}\, I_2 &\to I_1 /_{\mathit{Int}} I_2] \ \textbf{if}\ I_2 \neq 0 \\
\circ\, I_1 \,\texttt{<=}\, I_2 &\to I_1 \leq_{\mathit{Int}} I_2 \\
\circ\, \texttt{not true} &\to \texttt{false} \\
\circ\, \texttt{not false} &\to \texttt{true} \\
\circ\, \texttt{true and}\, B_2 &\to B_2 \\
\circ\, \texttt{false and}\, B_2 &\to \texttt{false} \\
\circ\, \langle C, \sigma \rangle[X \,\texttt{:=}\, I] &\to \langle C, \sigma[I/X] \rangle[\,\texttt{skip}\,] \ \textbf{if}\ \sigma(X) \neq \bot \\
\circ\, \texttt{skip ;}\, S_2 &\to S_2 \\
\circ\, \texttt{if true then}\, S_1 \,\texttt{else}\, S_2 &\to S_1 \\
\circ\, \texttt{if false then}\, S_1 \,\texttt{else}\, S_2 &\to S_2 \\
\circ\, \texttt{while}\, B \,\texttt{do}\, S &\to \texttt{if}\, B \,\texttt{then}\, (S \,\texttt{;}\, \texttt{while}\, B \,\texttt{do}\, S) \,\texttt{else skip} \\
\circ\, \langle\, \texttt{var}\, \mathit{Xl} \,\texttt{;}\, S \rangle &\to \langle S, (\mathit{Xl} \mapsto 0) \rangle
\end{aligned}
$$

Figure 3.40: $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ — rewriting logic theory corresponding to the third embedding of the reduction semantics with evaluation contexts of IMP.

all the left-hand-side terms of reduction rules of IMP in Figure 3.31 that contain a subterm in contextual representation contain that term at the top. As already discussed, if one preferred to write, e.g., the lookup RSEC rule as $\langle c[x], \sigma \rangle \to \langle c[\sigma(x)], \sigma \rangle$ if $\sigma(x) \neq \bot$, then one would need an additional generic rule, namely $\circ\, \langle S, \sigma \rangle \to \mathit{Cfg}'$ **if** $\circ\, \langle \mathit{split}(S), \sigma \rangle \to \mathit{Cfg}'$. While these generic rules take care of splitting and can be generated relatively automatically, the remaining rewrite rules that correspond to the reduction rules still make explicit use of the internal (to the embedding) *plug* operation, which can arguably be perceived by language designers as an inconvenience.

Figure 3.40 shows the rewriting logic theory $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ obtained by applying our third embedding (shown in Figure 3.35). These rules are also added to those corresponding to evaluation contexts in Figure 3.37 and, like in the second embedding, there is precisely one unconditional rewrite rule corresponding to each RSEC rule of IMP. We also need to add a generic conditional rule, the first one, which completely encapsulates the rewriting logic representation of the splitting/plugging mechanism, so that the language designer can next focus exclusively on the semantic rules rather than on their representation in rewriting logic. The second rewrite rule in Figure 3.40 corresponds to the characteristic rule of reduction semantics with evaluation contexts and, as discussed, it is optional; if one includes it, as we did, we think that its definition in Figure 3.40 is as simple and natural as it can be. In what regards the remaining rewrite rules, the only perceivable difference between them and their corresponding reduction rules is that they are preceded by $\circ$.

All the above suggest that, in spite of its apparently advanced context-sensitivity and splitting/plugging mechanism, reduction semantics with evaluation contexts can be safely regarded as a methodological fragment of rewriting logic. Or, put differently, while context-sensitive reduction seems crucial for programming language semantics, it is in fact unnecessary. A conditional rewrite framework can methodologically achieve the same results, and as discussed in this chapter, so can do for the other conventional language semantics approaches.

The following corollary of Theorems 7, 8, and 9 establishes the faithfulness of the representations of the reduction semantics with evaluation contexts of IMP in rewriting logic:

**Corollary 5.** *For any* IMP *configurations* $C$ *and* $C'$*, the following equivalences hold:*

$$\begin{aligned} \text{RSEC(IMP)} \vdash C \to C' &\iff \mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}} \vdash \circ\,\overline{C} \to \overline{C'} \\ &\iff \mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}} \vdash \circ\,\overline{C} \to \overline{C'} \\ &\iff \mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}} \vdash \circ\,\overline{C} \to \overline{C'} \end{aligned}$$

*and*

$$\begin{aligned} \text{RSEC(IMP)} \vdash C \to^\star C' &\iff \mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}} \vdash \star\,\overline{C} \to \star\,\overline{C'} \\ &\iff \mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}} \vdash \star\,\overline{C} \to \star\,\overline{C'} \\ &\iff \mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}} \vdash \star\,\overline{C} \to \star\,\overline{C'} \end{aligned}$$

Therefore, there is no perceivable computational difference between the reduction semantics with evaluation contexts RSEC(IMP) and its corresponding rewriting logic theories.

### ☆ Reduction Semantics with Evaluation Contexts of IMP in Maude

Figure 3.41 shows a Maude representation of the rewrite theory $\mathcal{R}^{\Box}_{\text{RSEC(IMP)}}$ in Figure 3.37 that embeds IMP's evaluation contexts by making explicit the split/plug mechanism which is implicit in reduction semantics with evaluation contexts. Figure 3.41 also includes the Maude definition of configurations (see Figure 3.36). We took the freedom to implement a simple optimization which works well in Maude, but which may not work as well in other engines or systems (which is why we did not incorporate it as part of the general procedure to represent reduction semantics with evaluation contexts in rewriting logic): we defined the contextual representation operation `_[_]` to have as result the *kind* (see Section 2.7) `[Syntax]` instead of the sort `Syntax`. This allows us to include the equation `plug(Syn) = Syn`, where `Syn` is a variable of *sort* `Syntax`, which gives us the possibility to also use terms which do not make use of contexts in the right-hand sides of rewrite rules. To test the rules for splitting, one can write Maude commands such as the one below, asking Maude to search for all splits of a given term:

```
search split(3 <= (2 + X) / 7) =>! Syn:[Syntax] .
```

The `!` tag on the arrow `=>` in the command above tells Maude to only report the normal forms, in this case the completed splits. As expected, Maude finds all seven splits and outputs the following:

```
Solution 1 (state 1)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> [][3 <= (2 + X) / 7]

Solution 2 (state 2)
states: 8  rewrites: 19 in ... cpu (. real) (0 rewrites/second)
Syn --> ([] <= (2 + X) / 7)[3]

Solution 3 (state 3)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= [])[(2 + X) / 7]

Solution 4 (state 4)
```

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

mod IMP-SPLIT-PLUG-EVALUATION-CONTEXTS is including IMP-CONFIGURATIONS-EVALUATION-CONTEXTS .
  sorts Syntax Context .  subsorts AExp BExp Stmt Configuration < Syntax .
  op [] : -> Context .  op _[_] : Context Syntax -> [Syntax] [prec 1] .
  ops split plug : Syntax -> Syntax . --- to split Syntax into context[redex]

  var X : Id .  var A A1 A2 : AExp .  var B B1 B2 : BExp .  var S S1 S2 : Stmt .
  var Sigma : State .  var I1 : Int .  var Syn : Syntax .  var C : Context .

  rl split(Syn) => [][Syn] .  eq plug([][Syn]) = Syn .  eq plug(Syn) = Syn .

  op <_,_> : Context State -> Context .  eq plug(< C, Sigma > [Syn]) = < plug(C[Syn]), Sigma > .
 crl split(< S, Sigma >) => < C, Sigma > [Syn] if split(S) => C[Syn] .

  op _+_ : Context AExp -> Context .  eq plug((C + A2)[Syn]) = plug(C[Syn]) + A2 .
 crl split(A1 + A2) => (C + A2)[Syn] if split(A1) => C[Syn] .
  op _+_ : AExp Context -> Context .  eq plug((A1 + C)[Syn]) = A1 + plug(C[Syn]) .
 crl split(A1 + A2) => (A1 + C)[Syn] if split(A2) => C[Syn] .

  op _/_ : Context AExp -> Context .  eq plug((C / A2)[Syn]) = plug(C[Syn]) / A2 .
 crl split(A1 / A2) => (C / A2)[Syn] if split(A1) => C[Syn] .
  op _/_ : AExp Context -> Context .  eq plug((A1 / C)[Syn]) = A1 / plug(C[Syn]) .
 crl split(A1 / A2) => (A1 / C)[Syn] if split(A2) => C[Syn] .

  op _<=_ : Context AExp -> Context .  eq plug((C <= A2)[Syn]) = plug(C[Syn]) <= A2 .
 crl split(A1 <= A2) => (C <= A2)[Syn] if split(A1) => C[Syn] .
  op _<=_ : Int Context -> Context .  eq plug((I1 <= C)[Syn]) = I1 <= plug(C[Syn]) .
 crl split(I1 <= A2) => (I1 <= C)[Syn] if split(A2) => C[Syn] .

  op not_ : Context -> Context .  eq plug((not C)[Syn]) = not plug(C[Syn]) .
 crl split(not B) => (not C)[Syn] if split(B) => C[Syn] .

  op _and_ : Context BExp -> Context .  eq plug((C and B2)[Syn]) = plug(C[Syn]) and B2 .
 crl split(B1 and B2) => (C and B2)[Syn] if split(B1) => C[Syn] .

  op _:=_ : Id Context -> Context .  eq plug((X := C)[Syn]) = X := plug(C[Syn]) .
 crl split(X := A) => (X := C)[Syn] if split(A) => C[Syn] .

  op _;_ : Context Stmt -> Context .  eq plug((C ; S2)[Syn]) = plug(C[Syn]) ; S2 .
 crl split(S1 ; S2) => (C ; S2)[Syn] if split(S1) => C[Syn] .

  op if_then_else_ : Context Stmt Stmt -> Context .
 crl split(if B then S1 else S2) => (if C then S1 else S2)[Syn] if split(B) => C[Syn] .
  eq plug((if C then S1 else S2)[Syn]) = if plug(C[Syn]) then S1 else S2 .
endm
```

Figure 3.41: The configuration and evaluation contexts of IMP in Maude, as needed for the three variants of reduction semantics with evaluation contexts of IMP in Maude.

```
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= ([] / 7))[2 + X]

Solution 5 (state 5)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= (([] + X) / 7))[2]

Solution 6 (state 6)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= ((2 + []) / 7))[X]

Solution 7 (state 7)
states: 8  rewrites: 19 in ... cpu (... real) (0 rewrites/second)
Syn --> (3 <= 2 + X / [])[7]
```

If, however, one replaces any of the rules for splitting with equations, then, as expected, one looses some of the splitting behaviors. For example, if one replaces the generic rule for splitting `rl split(Syn) => [][Syn]` by an apparently equivalent equation `eq split(Syn) = [][Syn]`, then Maude will be able to detect no other splitting of a term $t$ except for $\Box[t]$ (because Maude executes the equations before the rules; see Section 2.8).

Figure 3.42 shows two Maude modules implementing the first two rewriting logic theories $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ (Figure 3.38) and $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ (Figure 3.39), and Figure 3.43 shows the Maude module implementing the third rewrite theory $\mathcal{R}^{\boxed{3}}_{\text{RSEC(IMP)}}$ (Figure 3.40), respectively. Each of these three Maude modules imports the module `IMP-CONFIGURATION-EVALUATION-CONTEXTS` defined in Figure 3.41 and is executable. Maude, through its rewriting capabilities, therefore yields an IMP reduction semantics with evaluation contexts interpreter for each of the three modules in Figures 3.42 and 3.43. For any of them, 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: 39356 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < skip, n |-> 0 & s |-> 5050 >
state = (n |-> 0 , s |-> 5050) >
```

One can use any of the general-purpose tools provided by Maude on the reduction semantics with evaluation contexts definitions 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. Not unexpectedly, the same number of states as in the case of small-step SOS and MSOS will be discovered by this search command, namely 1509. Indeed, the splitting/cooling mechanism of RSEC is just another way to find where the next reduction step should take place; it does not generate any different reductions of the original configuration.

### 3.7.4 Notes

Reduction semantics with evaluation contexts was introduced by Felleisen and his collaborators (see, e.g., [29, 99]) as a variant small-step structural operational semantics. By making the evaluation

```
mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS .
  var X : Id .  var I I1 I2 : Int .  var B B2 : BExp .  var S S1 S2 : Stmt .
  var Xl : List{Id} .  var Sigma : State .  var Cfg : Configuration .  var C : Context .

 crl o Cfg => plug(< C,Sigma >[Sigma(X)]) if split(Cfg) => < C,Sigma >[X]
  /\ Sigma(X) =/=Bool undefined .
 crl o Cfg => plug(C[I1 +Int I2]) if split(Cfg) => C[I1 + I2] .
 crl o Cfg => plug(C[I1 /Int I2]) if split(Cfg) => C[I1 / I2]
  /\ I2 =/=Bool 0 .
 crl o Cfg => plug(C[I1 <=Int I2]) if split(Cfg) => C[I1 <= I2] .
 crl o Cfg => plug(C[false]) if split(Cfg) => C[not true] .
 crl o Cfg => plug(C[true]) if split(Cfg) => C[not false] .
 crl o Cfg => plug(C[B2]) if split(Cfg) => C[true and B2] .
 crl o Cfg => plug(C[false]) if split(Cfg) => C[false and B2] .
 crl o Cfg => plug(< C,Sigma[I / X] >[skip]) if split(Cfg) => < C,Sigma >[X := I]
  /\ Sigma(X) =/=Bool undefined .
 crl o Cfg => plug(C[S2]) if split(Cfg) => C[skip ; S2] .
 crl o Cfg => plug(C[S1]) if split(Cfg) => C[if true  then S1 else S2] .
 crl o Cfg => plug(C[S2]) if split(Cfg) => C[if false then S1 else S2] .
 crl o Cfg => plug(C[if B then (S ; while B do S) else skip]) if split(Cfg) => C[while B do S] .
  rl o < var Xl ; S > => < S,(Xl |-> 0) > .
endm



mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS .
  var X : Id .  var I I1 I2 : Int .  var B B2 : BExp .  var S S1 S2 : Stmt .
  var Xl : List{Id} .  var Sigma : State .  var Cfg Cfg' : Configuration .  var C : Context .

 crl o Cfg => Cfg' if o split(Cfg) => Cfg' .   --- generic rule enabling splitting

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

Figure 3.42: The first two reduction semantics with evaluation contexts of IMP in Maude.

```
mod IMP-SEMANTICS-EVALUATION-CONTEXTS is including IMP-SPLIT-PLUG-EVALUATION-CONTEXTS .
  var X : Id .  var I I1 I2 : Int .  var B B2 : BExp .  var S S1 S2 : Stmt .  var Xl : List{Id} .
  var Sigma : State .  var Cfg Cfg' : Configuration .  var Syn Syn' : Syntax  .  var C : Context .

  crl o Cfg => Cfg' if plug(o split(Cfg)) => Cfg' .           --- generic rule enabling splitting
  crl o C[Syn] => C[Syn'] if C =/=Bool [] /\ o Syn => Syn' .  --- characteristic rule

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

Figure 3.43: The third reduction semantics with evaluation contexts of IMP in Maude.

context explicit and modifiable, reduction semantics with evaluation contexts is considered by many to be a significant improvement over small-step SOS. Like small-step SOS, reduction semantics with evaluation contexts has been broadly used to give semantics to programming languages and to various calculi. We here only briefly mention some strictly related work.

How expensive is the splitting of a term into an evaluation context and a redex? Unfortunately, it cannot be more efficient than testing the membership of a word to a context-free grammar and the latter is expected to be cubic in the size of the original term (folklore). Indeed, consider $G$ an arbitrary CFG whose start symbol is $S$ and let $G_C$ be the "evaluation context" CFG grammar adding a fresh "context" nonterminal $C$, a fresh terminal $\#$, and productions $C \to \Box \mid CS\#$. Then it is easy to see that a word $\alpha$ is in the language of $G$ if and only if $\#\alpha\#$ can be split as a contextual representation (can only be $(\Box\alpha\#)[\#]$). Thus, we should expect, in the worst case, a *cubic* complexity to split a term into an evaluation context and a redex. An additional exponent needs to be added, thus making splitting expected to be a *quadratic* operation in the worst case, when nested contexts are allowed in rules (i.e., when the redex is itself a contextual representation). Unfortunately, this terrible complexity needs to be paid at each step of reduction, not to mention that the size of the program to reduce can also grow as it is reduced. One possibility to decrease this complexity is to attempt to incrementally compute at each step the evaluation context that is needed at the next step (like in refocusing; see below); however, in the worst case the right-hand sides of rules may contain no contexts, in which case a fresh split is necessary at each step.

Besides our own efforts, we are aware of three other attempts to develop executable engines for reduction semantics with evaluation contexts, which we discuss here in chronological order:

1. A specification language for syntactic theories with evaluation contexts is proposed by Xiao *et al.* [101, 100], together with a system which generates Ocaml interpreters from specifications. Although the compiler in [101, 100] is carefully engineered, as rightfully noticed by Danvy and Nielsen in [23] it cannot avoid the quadratic overhead due to the context-decomposition step. This is consistent with our own observations expressed at several places in this section, namely that the advanced parsing underlying reduction semantics with evaluation contexts is the most expensive part when one is concerned with execution. Fortunately, the splitting of syntax into context and redex can be and typically is taken for granted in theoretical developments, making abstraction of the complexity of its implementation.

2. A technique called *refocusing* is proposed by Danvy and Nielsen in [23, 22]. The idea underlying refocusing is to keep the program decomposed at all times (in a first-order continuation-like form) and to perform minimal changes to the resulting structure to find the next redex. Unfortunately, refocusing appears to work well only with restricted RSEC definitions, namely ones whose evaluation contexts grammar has the property of unique decomposition of a term into a context and a redex (so constructs like the non-deterministic addition of IMP are disallowed), and whose reduction rules are deterministic.

3. PLT-Redex, which is implemented in Scheme by Findler and his collaborators [43, 28], is perhaps the most advanced tool developed specifically to execute reduction semantics with evaluation contexts. PLT-Redex builds upon a direct implementation of context-sensitive reduction, so it cannot avoid the worst-case quadratic complexity of context decomposition, same as the interpreters generated by the system in [101, 100] discussed above. Several large language semantics engineering case studies using PLT-Redex are discussed in [28].

Our embeddings of reduction semantics with evaluation contexts into rewriting logic are inspired from a related embedding by Șerbănuță *et al.* in [85]. The embedding in [85] was similar to our third embedding here, but it included splitting rules also for terms in reducible form, e.g., $split(I_1 <= I_2) \rightarrow \Box[I_1 <= I_2]$. Instead, we preferred to include a generic rule $split(Syn) \rightarrow \Box[Syn]$ here, which allows us to more mechanically derive the rewrite rules for splitting from the CFG of evaluation contexts. Calculating the exact complexity of our approach seems to be hard, mainly because of optimizations employed by rewrite engines, e.g., indexing. Since at each step we still search for all the relevant splits of the term into an evaluation context and a redex, in the worst case we still pay the quadratic complexity. However, as suggested by the performance numbers in [85] comparing Maude running the resulting rewrite theory against PLT-Redex, which favor the former by a large margin, our embeddings may serve as alternative means to getting more efficient implementations of reduction semantics engines. There are strong reasons to believe that our third embedding can easily be automated in a way that the user never sees the split/plug operations.

### 3.7.5 Exercises

**Exercise 108.** *Suppose that one does not like mixing semantic components with syntactic evaluation contexts as we did above (by including the production* $Context ::= \langle Context, State \rangle$*). Instead, suppose that one prefers to work with configuration tuples like in SOS, holding the various components needed for the language semantics, the program or fragment of program being just one of them. In other words, suppose that one wants to make use of the contextual representation notation only on the*

*syntactic component of configurations. In this case, the characteristic rule becomes*

$$\frac{\langle e, \gamma\rangle \to \langle e', \gamma'\rangle}{\langle c[e], \gamma\rangle \to \langle c[e'], \gamma'\rangle}$$

*where $\gamma$ and $\gamma'$ consist of configuration semantic components that are necessary to evaluate $e$ and $e'$, respectively, such as states, outputs, stacks, etc. Modify accordingly the six reduction semantics with evaluation contexts rules discussed at the beginning of Section 3.7.*

*The advantage of this approach is that it allows the evaluation contexts to be defined exclusively over the syntax of the language. However, configurations holding code and state still need to be defined. Moreover, many rules which looked compact before, such as $i_1$ `<=` $i_2 \to i_1 \leq_{Int} i_2$, will now look heavier, e.g., $\langle i_1$ `<=` $i_2, \sigma\rangle \to \langle i_1 \leq_{Int} i_2, \sigma\rangle$.*

**Exercise 109.** *Like in Exercise 108, suppose that one does not like to mix syntactic and semantic components in evaluation contexts, but that, instead, one is willing to accept to slightly enrich the syntax of the programming language with a special statement construct Stmt ::=* `var` *Id* `=` *AExp which both declares and initializes a variable*[7]*. Then*

1. *Write structural identities that desugar the current top-level program variable declarations* `var` $x_1, ..., x_n$ `;` $s$ *into statements of the form* `var` $x_1$ `=` $0$ `;` ... `;` `var` $x_n$ `=` $0$ `;` $s$.
2. *Add a new context production that allows evaluation after the new variable declarations.*
3. *Modify the variable lookup and assignment rules discussed above so that one uses the new declarations instead of a state. Hint: the context should have the form* `var` $x$ `=` $i$ `;` $c$.

*The advantage of this approach is that one does not need an explicit state anymore, so the resulting definition is purely syntactic. In fact, the state is there anyway, but encoded syntactically as a sequence of variable initializations preceding any other statement. This trick works in this case, but it cannot be used as a general principle to eliminate configurations in complex languages.*

**Exercise$^\star$ 110.** *Exercise 108 suggests that one can combine MSOS (Section 3.6) and evaluation contexts, in that one can use MSOS's labels to obtain modularity at the configuration level and one can use the evaluation contexts idea to detect and modify the contexts/redexes in the syntactic component of a configuration. Rewrite the six rules discussed at the beginning of Section 3.7 as they would appear in a hypothetical framework merging MSOS and evaluation contexts.*

**Exercise 111.** *Modify the reduction semantics with evaluation contexts of* IMP *in Figures 3.30 and 3.31 so that* `/` *short-circuits when its numerator evaluates to* $0$.
*Hint: Make* `/` *strict in only the first argument, then use a rule to reduce* $0$`/`$a_2$ *to 0 and a rule to reduce* $i_1$`/`$a_2$ *to* $i_1$`/'`$a_2$ *when* $i_1 \neq 0$, *where* `/'` *is strict in its second argument, and finally a rule to reduce* $i_1$`/'`$i_2$ *to* $i_1 /_{Int} i_2$ *when* $i_2 \neq 0$.

**Exercise 112.** *Modify the reduction semantics with evaluation contexts of* IMP *in Figures 3.30 and 3.31 so that conjunction is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments.*

[7] Similar language constructs exist in many programming language (C, Java, etc.), so one may find their inclusion in the language acceptable.

**Exercise 113.** *Give an alternative reduction semantics of* IMP *with evaluation contexts following the approach in Exercise 108 (that is, use evaluation contexts only for the* IMP *language syntax, and handle the semantic components using configurations, like in SOS).*

**Exercise 114.** *Give an alternative reduction semantics of* IMP *with evaluation contexts following the approach in Exercise 109.*

**Exercise$^\star$ 115.** *Give a semantics of* IMP *using the hypothetical framework combining reduction semantics with evaluation contexts and MSOS proposed in Exercise 110.*

**Exercise 116.** *Modify the rewrite theory* $\mathcal{R}^{\square}_{\text{RSEC(IMP)}}$ *in Figure 3.37 so that later on one can define the reduction semantics of* / *to short-circuit when the numerator evaluates to* 0 *(as required in Exercises 118, 124, and 130).*

**Exercise 117.** *Modify the rewrite theory* $\mathcal{R}^{\square}_{\text{RSEC(IMP)}}$ *in Figure 3.37 so that one can later on define the reduction semantics of conjunction to be non-deterministically strict in both its arguments (as required in Exercises 119, 125, and 131).*

**Exercise 118.** *Modify the rewrite theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the reduction semantics of* / *that short-circuits when the numerator evaluates to* 0 *(see also Exercise 116).*

**Exercise 119.** *Modify the rewrite theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the reduction semantics of conjunction that defines it as non-deterministically strict in both its arguments (see also Exercise 117).*

**Exercise 120.** *As discussed in several places so far in Section 3.7, the reduction semantics rules for variable lookup and assignment can also be given in a way in which their left-hand-side terms are not in contextual representation (i.e.,* $\langle c[x], \sigma \rangle$ *instead of* $\langle c, \sigma \rangle[x]$*, etc.). Modify the corresponding rewrite rules of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for this alternative reduction semantics.*

**Exercise 121.** *Modify the rewriting logic theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the alternative reduction semantics with evaluation contexts of* IMP *in Exercise 113.*

**Exercise 122.** *Modify the rewriting logic theory* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$ *in Figure 3.38 to account for the alternative reduction semantics with evaluation contexts of* IMP *in Exercise 114.*

**Exercise$^\star$ 123.** *Combining the underlying ideas of the embedding of MSOS in rewriting logic discussed in Section 3.6.3 and the embedding of reduction semantics with evaluation contexts in Figure 3.33, give a rewriting logic semantics of* IMP *corresponding to the semantics of* IMP *in Exercise 115.*

**Exercise 124.** *Same as Exercise 118, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

**Exercise 125.** *Same as Exercise 119, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

**Exercise 126.** *Same as Exercise 120, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

**Exercise 127.** *Same as Exercise 121, but for* $\mathcal{R}^{\boxed{2}}_{\text{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\text{RSEC(IMP)}}$*).*

**Exercise 128.** *Same as Exercise 122, but for* $\mathcal{R}^{\boxed{2}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.39 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

**Exercise$^\star$ 129.** *Same as Exercise 123, but for Figure 3.39 (instead of Figure 3.38).*

**Exercise 130.** *Same as Exercise 118, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

**Exercise 131.** *Same as Exercise 119, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

**Exercise 132.** *Same as Exercise 120, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

**Exercise 133.** *Same as Exercise 121, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

**Exercise 134.** *Same as Exercise 122, but for* $\mathcal{R}^{\boxed{3}}_{\mathrm{RSEC(IMP)}}$ *in Figure 3.40 (instead of* $\mathcal{R}^{\boxed{1}}_{\mathrm{RSEC(IMP)}}$*).*

**Exercise$^\star$ 135.** *Same as Exercise 123, but for Figure 3.40 (instead of Figure 3.38).*

**Exercise 136.** *Modify the Maude code in Figures 3.41 and 3.42, 3.43 so that* / *short-circuits when its numerator evaluates to* 0 *(see also Exercises 111, 116, 118, 124, and 130).*

**Exercise 137.** *Modify the Maude code in Figures 3.41 and 3.42, 3.43 so that conjunction is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments (see also Exercises 112, 117, 119, 125, and 131).*

**Exercise 138.** *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the alternative reduction semantics in Exercises 120, 126, and 132.*

**Exercise 139.** *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the alternative reduction semantics in Exercises 121, 127, and 133.*

**Exercise 140.** *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the alternative reduction semantics in Exercises 122, 128, and 134.*

**Exercise$^\star$ 141.** *Modify the Maude code in Figures 3.41 and 3.42, 3.43 to account for the semantics in Exercises 123, 129, and 135.*

**Exercise 142.** *Same as Exercise 68, but for reduction semantics with evaluation contexts instead of small-step SOS: add variable increment to* IMP*, like in Section 3.7.2.*

**Exercise 143.** *Same as Exercise 72, but for reduction semantics with evaluation contexts instead of small-step SOS: add input/output to* IMP*, like in Section 3.7.2.*

**Exercise$^\star$ 144.** *Consider the hypothetical framework combining MSOS with reduction semantics with evaluation contexts proposed in Exercise 110, and in particular the* IMP *semantics in such a framework in Exercise 115, its rewriting logic embeddings in Exercises 123, 129, and 135, and their Maude implementation in Exercise 141. Define the semantics of the input/output constructs above modularly first in the framework in discussion, then using the rewriting logic embeddings, and finally in Maude.*

**Exercise 145.** *Same as Exercise 77, but for reduction semantics with evaluation contexts instead of small-step SOS: add abrupt termination to* IMP*, like in Section 3.7.2.*

**Exercise 146.** *Same as Exercise 85, but for reduction semantics with evaluation contexts instead of small-step SOS: add dynamic threads to* IMP*, like in Section 3.7.2.*

**Exercise 147.** *Same as Exercise 90, but for reduction semantics with evaluation contexts instead of small-step SOS: add local variables using* `let` *to* IMP*, like in Section 3.7.2.*

**Exercise$^\star$ 148.** *This exercise asks to define* IMP++ *in reduction semantics, in various ways. Specifically, redo Exercises 95, 96, 97, 98, and 99, but for the reduction semantics with evaluation contexts of* IMP++ *discussed in Section 3.7.2 instead of its small-step SOS in Section 3.5.6.*