Universidade Federal Fluminense

FABRICIO CHALUB BARBOSA DO ROSÁRIO

An Implementation of Modular Structural Operational Semantics in Maude

NITERÓI

2005

Universidade Federal Fluminense

FABRICIO CHALUB BARBOSA DO ROSÁRIO

An Implementation of Modular Structural Operational Semantics in Maude

M. Sc. dissertation submitted to the Graduate School of Computation of the Fluminense Federal University as a partial requirement for the title of Master in Science. Area: Distributed and Parallel Processing/Formal Methods

Advisor:
Christiano de Oliveira Braga

NITERÓI

2005

An Implementation of Modular Structural Operational Semantics in Maude

Fabricio Chalub Barbosa do Rosário

M. Sc. dissertation submitted to the Graduate School of Computation of the Fluminense Federal University as a partial requirement for the title of Master in Science.

Committee:

Prof. Christiano de Oliveira Braga, D. Sc. / IC-UFF
(Advisor)

Prof. Edward Hermann Haeusler, D. Sc. / PUC-Rio

Prof. Peter D. Mosses, Ph. D. / Univ. of Wales Swansea

Niterói, May 27, 2005.

Abstract

This dissertation presents a formal tool for Modular Structural Operational Semantics (MSOS), based on the conversion from MSOS to Rewriting Logic recently developed by Braga and Meseguer. The implementation, named *Maude MSOS Tool* (MMT), was written in Maude, a high-performance implementation of Rewriting Logic. The development of MMT attempts not only to provide an MSOS interpreter that uses a specification language that is closer to the domain of MSOS specifications than to Maude specifications, but also to demonstrate what can be accomplished when one develops a formal tool in the Maude environment, since it allows the use of other formal tools already available with MSDF specifications. We have demonstrated this by simulating and model checking concurrent programs and distributed algorithms. Another aim is to provide an example of a non-trivial extension of Full Maude and to create a tool that is itself extensible.

Contents

1 Introduction **1**

2 Background **3**

2.1 Structural Operational Semantics and Modular Structural Operational Semantics . . . 3

2.1.1 Structural Operational Semantics . . . 3

2.1.2 Modular SOS . . . 6

2.2 Rewriting Logic . . . 9

2.3 Maude . . . 13

2.3.1 Tools . . . 19

2.3.1.1 Reducing and rewriting terms . . . 19

2.3.1.2 Searching for states . . . 20

2.3.1.3 Model checking specifications . . . 22

2.3.2 Metalevel programming . . . 25

2.3.2.1 Maude as meta-tool . . . 29

2.3.3 Input and output facilities . . . 30

2.3.4 Full Maude . . . 33

2.3.4.1 System and functional modules . . . 33

2.3.4.2 Theories, views, and parameterized modules. . . . 34

2.3.4.3 Extending Full Maude . . . 36

2.4 Modular Rewriting Semantics . . . 38

2.4.1 Modular Rewriting Semantics and MSOS . . . 44

3 Related work **48**

4 Maude MSOS Tool **52**

4.1 Notational conventions . . . 53

4.2 MSDF syntax . . . 53

4.2.1 Modules 54
4.2.2 Datatype definitions 55
4.2.3 Labels 57
4.2.4 Semantic transitions 58
4.3 Built-in operations on derived and parameterized sets 61
4.3.1 Sequences 62
4.3.2 Lists 62
4.3.3 Maps 62
4.3.4 Sets 63
4.4 User interface 63
4.5 A simple example 63

5 The implementation of MMT **66**
5.1 MMT as an extension of Full Maude 66
5.2 Modules 68
5.3 Datatypes 69
5.3.1 Compilation of type declarations 69
5.3.2 Compilation of typed syntactic trees 70
5.3.3 Compilation of functions 71
5.3.4 Compilation of module inclusion 71
5.3.5 Parameterized and derived types 73
5.3.5.1 View forwarding problem 74
5.3.5.2 Derived Sets 75
5.4 Processing label declarations 77
5.5 Processing MSOS transitions 79

6 Case studies **86**
6.1 Constructive MSOS 86
6.1.1 The CMSOS constructions 86
6.1.1.1 Expressions 87
6.1.1.2 Declarations 88
6.1.1.3 Abstractions 89
6.1.1.4 Commands 91

6.1.1.5 Concurrency . . . 92
6.1.2 ML . . . 92
6.1.2.1 Expressions . . . 93
6.1.2.2 Declarations . . . 94
6.1.2.3 Imperatives . . . 95
6.1.2.4 Abstractions . . . 96
6.1.2.5 Concurrency . . . 97
6.1.2.6 Example . . . 98
6.1.3 MiniJava . . . 100
6.1.3.1 Expressions . . . 101
6.1.3.2 Statements . . . 103
6.1.3.3 Classes . . . 103
6.1.3.4 Example . . . 104
6.2 Mini-Freja . . . 108
6.2.1 Abstract Syntax . . . 108
6.2.2 Semantics . . . 109
6.2.3 Example: sieve of Eratosthenes . . . 113
6.3 Distributed algorithms . . . 114
6.3.1 Process execution model . . . 115
6.3.1.1 Process communication models . . . 116
6.3.1.2 Justice . . . 117
6.3.2 Examples . . . 117
6.3.2.1 Another thread game . . . 117
6.3.2.2 Dining Philosophers . . . 119

7 Conclusion **123**
7.1 Design decisions and limitations . . . 123
7.1.1 Typed syntactic trees in conditions . . . 123
7.1.2 Limitations of the MSDF syntax in MMT . . . 123
7.1.3 Loading of modules . . . 125
7.1.4 Limitations on the generality of MSDF in MMT . . . 126
7.1.5 Automatic variables . . . 127
7.2 Enhancements to the tool — future work . . . 128

7.3 Contributions 130

Bibliography **131**

Appendix A – Constructive MSOS **137**

A.1 Expressions 137

A.2 Declarations 138

A.3 Commands 139

A.4 Abstractions 143

A.5 Concurrency 145

Appendix B – ML specification **150**

B.1 Expressions 150

B.2 Declarations 153

B.3 Imperatives 154

B.4 Abstractions 156

B.5 Concurrency 157

Appendix C – MiniJava specification **159**

C.1 Expressions 159

C.2 Statements 160

C.3 Classes 162

Appendix D – Mini-Freja specification **166**

Appendix E – Distributed algorithms **170**

E.1 Mutual exclusion using semaphores 170

E.2 Dining Philosophers 173

E.2.1 Remainder of the rules 173

E.2.2 Dining Philosophers, terminating specification 174

E.2.3 Fair scheduling 176

E.2.4 An incorrect specification 177

E.3 Bakery algorithm 179

E.4 Leader election on an asynchronous ring 185

Appendix F – Combinatory Logic in Maude **188**

Chapter 1

Introduction

Structural Operational Semantics (SOS), developed by Plotkin [58], is a well known framework commonly used on the formal specification of programming languages [61, 49] and concurrent systems [50, 51]. It is also widely used in formal semantics textbooks and lecture notes [29, 54, 62, 57, 58, 52]. From a software engineering perspective, however, SOS lacks an essential characteristic for the specification of complex systems: *modularity*. This has been solved by Mosses with the creation of Modular Structural Operational Semantics (MSOS) [53]. Recently, Mosses also developed a specification language for MSOS, the Modular SOS Specification Formalism (MSDF) [52].

Recent years have shown us a great deal of development of algebraic methods and especially their use as specification formalisms. Rewriting Logic [40] and Membership Equational Logic [41] are two notable examples that have been used to specify a wide variety of topics [17, 7, 18, 55, 6, 64, 44, 20, 59].

The relationship between Rewriting Logic and SOS [38, 6, 68, 44, 8], and in particular Modular SOS [8, 44] have been studied before.

This dissertation aims to close this loop by providing a formal environment for MSDF specifications, using the developed conversion from MSOS to Rewriting Logic. The implementation, named *Maude MSOS Tool* (MMT), was written in Maude 2.1.1 [14], a high-performance implementation of Rewriting Logic. The development of MMT attempts not only to provide an MSOS interpreter that uses a specification language that is closer to the domain of MSOS specifications than to Maude specifications, but also to demonstrate what can be accomplished when one develops a formal tool in the Maude environment, since it allows the use of other formal tools already available with MSDF specifications. We have demonstrated this by simulating and model checking concurrent programs and distributed algorithms. Another aim is to provide an example of a non-trivial extension of Full Maude and to create a tool that is itself extensible.

This dissertation is organized as follows. Chapter 2 gives the necessary background on all the frameworks used on the translation and implementation process; Chapter 3 shows other implementations of SOS and Modular SOS; Chapter 4 describes the syntax of MSDF, the specification language used by *Maude MSOS Tool*; Chapter 5 describes the implementation of *Maude MSOS Tool*; Chapter 6 shows several applications of MMT in the specification and verification of programming languages and distributed systems. Chapter 7 concludes this dissertation with some final remarks. The appendices have

additional material that were omitted from certain sections, specially in Chapter 6 for the sake of brevity.

Chapter 2

Background

This Chapter provides the background material about Structural Operational Semantics [58] and Modular SOS [53] (Section 2.1), Rewriting Logic [40] (Section 2.2), and its implementation engine Maude [14] (Section 2.3); the relationship between Modular SOS and Rewriting Logic is given by first introducing Modular Rewriting Semantics [8, 44] (Section 2.4) and then showing how to formalize Modular SOS with Modular Rewriting Semantics.

2.1 Structural Operational Semantics and Modular Structural Operational Semantics

2.1.1 Structural Operational Semantics

The Structural Operational Semantics (SOS) framework, defined by Plotkin in [58], is a commonly used framework for the definition of formal programming languages semantics [49] and concurrent systems [50, 51].

The operational semantics of a programming language in SOS is given by a labelled terminal transition system $(\Gamma, A, \rightarrow, T)$, where Γ is a set (of configurations $\gamma \in \Gamma$), A a set of labels, $\rightarrow \subseteq \Gamma \times A \times \Gamma$ is a ternary relation, and $T \subseteq \Gamma$ is the set of terminal configurations. SOS specifications are pairs (S, T), where S is the abstract syntax definition and T is the set of transitions. A transition $t \in T$ is specified using the notation $\gamma \xrightarrow{a} \gamma'$ which stands for $(\gamma, a, \gamma') \in \rightarrow$, that is, there is a transition from the configuration γ to the configuration γ', with label a. Conditional transitions are usually written as:

$$\frac{c_1, \ldots, c_n}{c}$$

where the conclusion c is a transition, and each condition c_i is either a transition or other type of conditions such as equations, set memberships, etc.

Configurations $\gamma \in \Gamma$ are tuples consisting of *value-added syntactic trees*, that is, syntactic trees in which branches may be final values, and any additional semantic com-

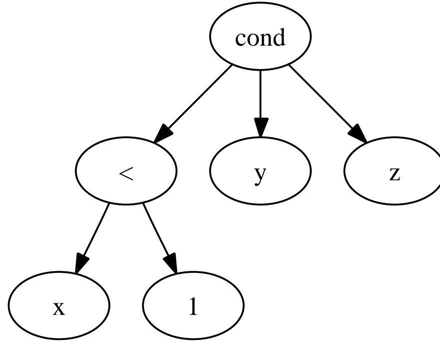

Figure 2.1: The value-added syntactic tree for `cond(x < 1, y, z)`

ponents, such as the bindings environment, stores, etc. We also use "term" to mean a value-added syntactic tree; figure 2.1 shows the tree for the term `cond(x < 1, y, z)`, according to some hypothetical grammar that specifies the syntax of a conditional construction in prefix format. The '`1`' should be seen as a final, computed, value. The use of abstract syntactic trees follows traditional practice of avoiding the complexity of concrete syntax that adds nothing to the precise understanding of the semantics of a programming language. Components are used to give semantics to (abstract) language constructs and are general auxiliary mathematical entities. For example, the bindings environment is usually modeled as a function that maps identifiers to values, for example, $\rho : \mathcal{I} \to \mathbb{N}$ so that we write $m = \rho(i)$, $i \in \mathcal{I}$, to access the value $m \in \mathbb{N}$ bound to the identifier i in the environment ρ. Other types of functions (or relations) may be defined as components.

Plotkin left open the problem of modularity in SOS specifications. For example, Rules 2.1 and 2.2 define the SOS of simple mathematical expressions. Let us introduce first the abstract syntax.

$$\begin{array}{lcl} m & \in & \mathbb{N} \\ e & \in & \mathsf{Exp} \\ & & e ::= m \mid e_0 + e_1 \end{array}$$

The transition rules give the meaning of the mathematical operation $e_0 + e_1$. First, e_0 is evaluated until it reaches a final value, a natural number (m_0), then the same for e_1 (m_1). Rule 2.2 rewrites $m_0 + m_1$ to the natural sum of m_0 and m_1.

$$\frac{e_0 \longrightarrow e_0'}{e_0 + e_1 \longrightarrow e_0' + e_1} \qquad \frac{e_1 \longrightarrow e_1'}{m_0 + e_1 \longrightarrow m_0 + e_1'} \tag{2.1}$$

$$m_0 + m_1 \longrightarrow m_0 + m_1 \tag{2.2}$$

The addition of bindings, for example, by the introduction of an ML-like '`let`' construct, requires the use of an environment component ($\rho \in \mathsf{Env}$) added to the configuration. In this simple specification, environments are finite functions from variables to final values Var $\to \mathbb{N}$.

We use here an example adapted from Plotkin's notes, which is a simpler form of the '`let`' construct from Standard ML [49]. Rule 2.3 specifies that, first, the expression e_0 is evaluated until a final value m is found. Rule 2.4 specifies that e_1 should be evaluated into e_1' in the context of a new environment, obtained by replacing all instances of the

variable x in the environment ρ by m and placing back the evaluated e_1' into the 'let' body. Finally, Rule 2.5 specifies that when e_1 is evaluated to a final value n, the entire expression should be replaced by n.

$$e ::= \texttt{let } x\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} \qquad x \in \text{Var} = \{x_1, x_2, \ldots\}$$

$$\frac{\rho \vdash e_0 \longrightarrow e_0'}{\rho \vdash \texttt{ let } x\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} \longrightarrow \texttt{ let } x\texttt{=}e_0' \texttt{ in } e_1 \texttt{ end}} \tag{2.3}$$

$$\frac{\rho[m/x] \vdash_{V \cup \{x\}} e_1 \longrightarrow e_1'}{\rho \vdash \texttt{ let } x\texttt{=}m \texttt{ in } e_1 \texttt{ end} \longrightarrow \texttt{ let } x\texttt{=}m \texttt{ in } e_1' \texttt{ end}} \tag{2.4}$$

$$\rho \vdash \texttt{ let } x\texttt{=}m \texttt{ in } n \texttt{ end} \longrightarrow n \tag{2.5}$$

Rule 2.6 locates the value of the variable x in the environment ρ

$$\frac{\rho \vdash m = \rho(x)}{\rho \vdash x \longrightarrow m} \tag{2.6}$$

Since now expressions are evaluated in the presence of an environment, the rules for mathematical expressions must be rewritten.

$$\frac{\rho \vdash e_0 \longrightarrow e_0'}{\rho \vdash e_0\texttt{+}e_1 \longrightarrow e_0'\texttt{+}e_1} \qquad \frac{\rho \vdash e_1 \longrightarrow e_1'}{\rho \vdash m_0\texttt{+}e_1 \longrightarrow m_0\texttt{+}e_1'} \tag{2.7}$$

$$\rho \vdash m_0\texttt{+}m_1 \longrightarrow m_0 + m_1 \tag{2.8}$$

Incidentally, the rules for arithmetic operations and the 'let' construction are in the so-called "small-step" operational semantics, since the evaluation of the value-added syntax tree is made one step at a time, substituting branches of the tree during the transition until a final value is (possibly) reached. An alternative to small-step semantics is the "big-step" semantics, in which the entire syntax tree is computed *directly* to a final value. The semantics of Standard ML is given in big-step style, for example [49]. Optionally, the big-step relation may use the symbol $\Rightarrow$ to distinguish from the "small-step" style. Let us exemplify by reinstating rules 2.1 and 2.2 together in rule 2.9 in big-step style:

$$\frac{e_0 \Rightarrow m_0 \quad e_1 \Rightarrow m_1}{e_0 + e_1 \Rightarrow m_0 + m_1} \tag{2.9}$$

The following rule is also necessary in big-step semantics.

$$n \Rightarrow n \tag{2.10}$$

SOS also has the concept of *operational conservative extensions* [1]. An extension of a set of rules is *operationally conservative* if provable transitions in the original system are the same as those in the extended system. It has been shown ([1]) that *source-dependent*

rules are a necessary condition for operational conservative extensions. A rule is source-dependent if all its variables are source-dependent. The source-dependent variables in a transition rule r are defined inductively as follows: (i) all variables in the source of r are source-dependent; if $t \to t'$ is a premise of r and all variables in t are source-dependent, then all variables in t' are source-dependent. To illustrate this, consider the following example, taken from [1, 27]. Consider two constants a and b, and a metavariable x. With the following rule alone, it is not possible to prove that $a \to a$.

$$\frac{x \to x}{a \to a}$$

However, if we extend the system by adding the following rule, it becomes possible to prove $a \to a$, by instantiating the metavariable x to the constant b. The problem is that the variable x is not source-dependent.

$$b \to b$$

2.1.2 Modular SOS

To solve the modularity problem in SOS, Mosses developed a framework called Modular SOS (MSOS) [53].[1] The key modularity point in MSOS lies on the *generalized transition systems* where the semantic components, such as the environment, are moved from the configurations to the transition label. The configurations rewritten by the transition rules consist only of value-added abstract syntax trees. Transition labels are understood as *arrows* of a category and adjacent labels in computations are required to be composable. Semantic components, now on the label, are referenced through *indices*. The idea is that a label may contain an unspecified number of components, but *only the components that are needed* in a particular transition must be made explicit. Formally a generalized transition system is a quadruple $(\Gamma, \mathbb{A}, \to, T)$ where $\mathbb{A}$ is a category with arrows A, such that $(\Gamma, A, \to, T)$ is a labelled terminal transition system. Computation requires that whenever a transition with label α is followed by a transition labelled α', it is required that α and α' are composable in $\mathbb{A}$ [53].

Since labels are now arrows of a category, let us discuss how *label categories* are used to model information that is processed in MSOS transitions. First, let us recall briefly that a category consists of: (i) a set of objects O; (ii) a set of arrows A; (iii) functions *source* and *target* from A to O; a partial function from $A \times A$ to A for composing arrows; and (iv) a function from O to A giving an identity arrow for each object. Labels which are identity arrows stand for *unobservable* transitions, which we discuss later in this Section. Normally three different types of label categories are used on MSOS specifications:

- the *discrete category*, with a single, identity, arrow for each object. These labels represent information that can be read by a transition but not written to, as is the case of *environments* (read-only information);
- the cartesian product of the sets of objects O and the set of arrows $O \times O$, with arrow (o, o') going from o to o'. Composition in this category eliminates intermediate

[1] The description of MSOS in this Section follows the description given by Mosses in [53, 52].

objects and identity arrows are of the form (o, o); usually this category is used to model information that may be read and changed by a transition, as is the case of *stores* (read-write information);

- the 1-object category where the set of arrows is O^*, the monoid of sequences generated by O. The identity arrow is the empty sequence (ε), and composition of arrows is sequence concatenation (given by the binary operation $\cdot$); it is used to model information that is *emitted* or *produced* by a transition, such as signaling an exception or outputting a value (write-only information).

The same considerations for operational conservative extensions are also applicable to MSOS with the notion of source-dependability extended to metavariables appearing on labels.

Now let us describe MSOS specifications, which are triples (S, L, T), where S is, as in SOS, the abstract syntax, L is the *label composition* specified as a *product* of the three categories described above, and T the usual set of transitions. Let us proceed now with the intuitive understanding of label expressions in MSOS, described at the beginning of this Section (the complete categorical aspects of MSOS is given in [53]). In MSOS, as we mentioned, components are accessed in labels through indices and can be of three different types, which ultimately reflects their categorical formalization: read-only, read-write, and write-only components.

MSOS defines a notation for the indices of each different type of component: a single, unprimed index i is associated with a read-only component, which is the same at the start and at the end of the transition; a pair of unprimed and primed indices i, i' is associated with a read-write component: the unprimed index refers to information present at the start of the transition and the primed index refers to information present at the end of the transition; a single, primed, index is associated with a write-only components and refers to information that is present at the end of the transition.

The different type of components have different requirement for the composability of adjacent labels. Two labels (L_1, L_2) are composable if and only if L_1 and L_2 have the same set of indices, and for each index i:

- if i indexes a read-only component, $L_1.i = L_2.i$;
- if i, i' indexes a read-write components, $L_1.i' = L_2.i$.

Write-only components do not affect the composability of labels.

The result of the composition of labels $L_1; L_2$, is determined by the composition of each pair index-component (i, c) in a label L (also called a *field*) pairwise joined by their respective indices, as follows:

- for read-only indices $(i, c); (i, c) = (i, c)$, the components are the same;
- for read-write indices $(i, c, i', c'); (i, c', i', c'') = (i, c, i', c'')$, that is, the composition eliminates intermediate components;

- for write-only indices $(i', c); (i', c') = (i', c \cdot c')$, that is, composition is given in terms of the binary operation of the monoid generated by i'.

Labels in MSOS may be classified as *unobservable* when read-write components do not change, and no new information is produced by write-only components. That is L is an unobservable label if and only if, for each index i:

- if i and i' index a read-write component, $L.i = L.i'$;
- if i' indexes a write-only component, $L.i' = \varepsilon$.

A transition rule, in its unconditional case, is written as $t \xrightarrow{\alpha} t'$ and specifies a triple relation between the terms t, and t', and the label α. Conditional transitions are written as in SOS:

$$\frac{c_1, \ldots, c_n}{c}$$

where it specifies that, if the conditions $c_1, \ldots, c_n$ hold, then the conclusion c also holds.

The label expression α uses a notation that is reminiscent of Standard ML notation for record patterns. Each field is written as $i = c$, with i the index and c the component, and the "rest of the label," is written with the notation '$\ldots$'. For example, a label α and an index ρ such that $\alpha.\rho = \rho_0$ is written as $\{\rho = \rho_0, \ldots\}$; a label α with indices σ and σ' such that $\alpha.\sigma = \sigma_0$ and $\alpha.\sigma' = \sigma_1$ is written as $\{\sigma = \sigma_0, \sigma' = \sigma_1, \ldots\}$; and a label α with an index τ' such that $\alpha.\tau' = \tau_0$ is written as $\{\tau' = \tau_0, \ldots\}$. The metavariable X ranges over arbitrary labels, and the metavariable U ranges over unobservable labels. Optionally, instead of writing $t \xrightarrow{U} t'$, one may write $t \longrightarrow t'$.

Let us illustrate MSOS revisiting the specifications for arithmetic expressions and introducing the rules for '`let`' expressions. Rules 2.11 and 2.12 specify the evaluation of expressions in MSOS.

$$\frac{e_0 \xrightarrow{X} e_0'}{e_0 + e_1 \xrightarrow{X} e_0' + e_1} \qquad \frac{e_1 \xrightarrow{X} e_1'}{m_0 + e_1 \xrightarrow{X} m_0 + e_1'} \tag{2.11}$$

$$m_0 + m_1 \longrightarrow m_0 + m_1 \tag{2.12}$$

To give semantics to a '`let`' expression, we add an environment to the specification by means of an index declaration in the labels. Rules 2.13, 2.14, and 2.15 specify in MSOS the meaning of '`let`' expressions. The informal description of Rule 2.14 is: to evaluate the e_1 expression inside the '`let`', evaluate one step of e_1 in the context of a new env-indexed component $(\rho[m/x])$ into e_1'; any changes to read-write components and any produced information by write-only components (represented by the notation "$\ldots$") should be carried onto the conclusion.

$$\frac{e_0 \xrightarrow{X} e_0'}{\texttt{let } x = e_0 \texttt{ in } e_1 \texttt{ end} \xrightarrow{X} \texttt{let } x = e_0' \texttt{ in } e_1 \texttt{ end}} \tag{2.13}$$

$$\frac{e_1 \xrightarrow{\{env = \rho[m/x], \ldots\}} e_1'}{\texttt{let } x = m \texttt{ in } e_1 \texttt{ end} \xrightarrow{\{env = \rho, \ldots\}} \texttt{let } x{=}m \texttt{ in } e_1' \texttt{ end}} \tag{2.14}$$

$$\texttt{let } x{=}m \texttt{ in } n \texttt{ end} \xrightarrow{U} n \tag{2.15}$$

Finally, rule 2.16 is analogous to rule 2.6. We use an unobservable label, represented by the metavariable U, and accessing the environment indexed by ρ with the notation $U.\rho$.

$$\frac{n = U.\rho(x)}{x \xrightarrow{U} n} \tag{2.16}$$

The transitions in MSOS may optionally operate on *typed* value-added syntactic trees. The type of the right-hand side is assumed to be the same as the type of the left-hand side. This kind of syntactic tree has one additional constraint that it specifies, of course, the *type* of the term to be matched against. Recall that, with a subset inclusion relation between sets, a term may be part of several different sets—an obvious example is the term '`100`', which is part of, say, the sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{Q}$. With this additional constraint, one may create specific rules for *each* particular type that a term may have. For example: on a hypothetical specification in which there are distinct components for the bindings of values and the bindings of closures, one may create a rule that specifies that an identifier, when being evaluated on the context of a '`MathExpression`', should lookup its value on the environment that maps identifiers to values, but when being evaluated on the context of a '`FunctionCall`', should lookup the function body on the environment that maps identifiers to closures, and so on.

2.2 Rewriting Logic

Rewriting logic (RWL) [40] is a logic of *change* in which both static and dynamic aspects of a system may be specified. It is also a *logical framework* which can represent many different logics, languages, operational formalisms and models of computation [17, 7, 18, 55, 6, 64, 44, 20, 59]. The dynamic aspects are specified in rewriting logic itself, using *labelled conditional rewrite rules* and the static aspects are specified in an underlying equational logic. Rewriting logic has several high-performance implementations [5, 14, 22].

This Section describes formally Rewriting Logic along with its equational sub-logic, the Membership Equational Logic (MEL) [41]. Practical examples are shown in Section 2.3, where we describe our chosen RWL implementation, Maude [14].

The underlying equational logic that rewriting logic commonly uses is the *membership equational logic*. MEL is a generalization of order-sorted equational logic where each term belongs to a *kind*, and each kind k has an associated poset $(S_k, \leq)$ of sorts. This allows the representation of partiality by defining *error terms* as terms with *kinds*, but with no associated *sort*. A detailed example is given, using Maude syntax, in Section 2.3.

Formally,[2] a signature in MEL is a triple $\Omega = (K, \Sigma, S)$ where K is a set of *kinds*,

[2]The following description follows the presentation of MEL in [16] and [41].

Σ is a K-kinded signature $\{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$, and S is a pairwise disjoint family of *sorts* $S = \{S_k\}_{k\in K}$. Following usual notation, we write T_Σ the K-kinded algebra of ground Σ-terms, and by $T_{\Sigma(X)}$ the K-kinded algebra of Σ-terms on the K-kinded set of variables X.

The atomic formulae of MEL are either *equations* $t = t'$, where t and t' are Σ-terms of the same kinds, or *membership axioms* of the form $t : s$, where t has kind k and $s \in S_k$. Sentences in MEL are Horn clauses on these atomic formulae:

$$(\forall X)\, A_0 \Leftarrow A_1 \wedge \cdots \wedge A_n$$

where A_i is either an equation or a membership axiom, and each $x_j \in X$ is a K-kinded variable. A theory in MEL is a pair (Ω, E) where E is the set of sentences composed of conditional Church-Rosser, terminating, and sort-decreasing equations and conditional membership axioms over the signature Ω.

Given a MEL theory $T = (\Omega, E)$ we say that T entails a sentence φ, and write $T \vdash \varphi$, if and only if φ is obtained by finite application of the following rules of deduction.

- Reflexivity.

$$\frac{}{E \vdash (\forall X)\, t = t}$$

- Symmetry.

$$\frac{E \vdash (\forall X)\, t = t'}{E \vdash (\forall X)\, t' = t}$$

- Transitivity.

$$\frac{E \vdash (\forall X)\, t = t' \quad E \vdash (\forall X)\, t' = t''}{E \vdash (\forall X)\, t = t''}$$

- Congruence.

$$\frac{E \vdash (\forall X)\, t_1 = t'_1 \quad \cdots \quad E \vdash (\forall X)\, t_n = t'_n}{E \vdash (\forall X)\, f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}$$

- Membership.

$$\frac{E \vdash (\forall X)\, t = t' \quad E \vdash (\forall X)\, t : s}{E \vdash (\forall X)\, t' : s}$$

- *Modus ponens* for equations.[3] Given a sentence:

$$(\forall X)\, t = t' \Leftarrow u_1 = v_1 \wedge \cdots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \cdots \wedge w_m : s_m$$

[3] Usually the rules for *modus ponens* for equations and membership axioms are shown together; we opted for the separation into two different rules to avoid an unnecessarily complex rule.

in the set E of axioms, and given a K-kinded assignment $\theta : X \to T_\Sigma(Y)$ then, for $1 \leq i \leq n$ and $1 \leq j \leq m$, where from θ we may obtain its unique extension to a Σ-homomorphism $\overline{\theta} : T_\Sigma(X) \to T_\Sigma(Y)$ (see [41, Section 2] for more details on this).

$$\frac{E \vdash (\forall Y)\,\overline{\theta}(u_i) = \overline{\theta}(v_i) \quad E \vdash (\forall Y)\,\overline{\theta}(w_j) : s_j}{E \vdash (\forall X)\,\overline{\theta}(t) = \overline{\theta}(t')}$$

- *Modus ponens* for membership axioms. Given a sentence:

$$(\forall X)\, t : s \Leftarrow u_1 = v_1 \wedge \cdots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \cdots \wedge w_m : s_m$$

in the set E of axioms, and given a K-kinded assignment $\theta : X \to T_\Sigma(Y)$ then, for $1 \leq i \leq n$ and $1 \leq j \leq m$.

$$\frac{E \vdash (\forall Y)\,\overline{\theta}(u_i) = \overline{\theta}(v_i) \quad E \vdash (\forall Y)\,\overline{\theta}(w_j) : s_j}{E \vdash (\forall X)\,\overline{\theta}(t) : s}$$

As mentioned before, practical examples on the use of MEL are given in Section 2.3.

A *rewrite theory*[4] is a tuple $\mathcal{R} = (\Omega, E, R)$ where (Ω, E) is a MEL theory, as described above; R is a set of universally quantified labelled conditional rewrite rules of the form ([10, Section 1.1])

$$l : t \to t' \Leftarrow (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \to w'_k)$$

with the following deduction rules:

- Reflexivity.

$$\frac{}{(\forall X)\, t \to t}$$

- Transitivity.

$$\frac{(\forall X)\, t_1 \to t_2 \quad (\forall X)\, t_2 \to t_3}{(\forall X)\, t_1 \to t_3}$$

- Equality.

$$\frac{(\forall X)\, u \to v \quad E \vdash (\forall X)\, u = u' \quad E \vdash (\forall X)\, v = v'}{(\forall X)\, u' = v'}$$

- Congruence. For each $f : k_1 \cdots k_n \to k$ in Σ

[4]We use the original presentation of rewrite theories in rewriting logic and not the *generalized* rewrite theories defined by Bruni and Meseguer in [10] since the more general construction of *frozen* operators is not explored by *Maude MSOS Tool*.

$$\frac{(\forall X)\, t_{j_1} \to t'_{j_1} \quad \cdots \quad (\forall X)\, t_{j_m} \to t'_{j_m}}{(\forall X)\, f(t_1, \ldots, t_{j_1}, \ldots, t_{j_m}, \ldots, t_n) \to f(t_1, \ldots, t'_{j_1}, \ldots, t'_{j_m}, \ldots, t_n)}$$

- Nested replacement[5]. For finite substitutions $\theta, \theta' : X \to T_\Sigma(Y)$. Given a rewrite rule, with $1 \leq i \leq n$.

  $$(\forall X)\, l : t \to t' \Leftarrow \bigwedge_i t_i \to t'_i$$

  For $1 \leq i \leq n$ and $x \in X$:

  $$\frac{(\forall Y)\, \theta(t_i) \to \theta(t'_i) \quad (\forall Y)\, \theta(x) \to \theta'(x)}{(\forall Y)\, \theta(t) \to \theta'(t)'}$$

  This rule means that, given a rule $r \in R$ and two substitutions θ, θ' for its variables such that for each $x \in X$ we have $\theta(x) \to \theta'(x)$, then r can be concurrently applied to the rewrites of its arguments, once that the conditions of r can be satisfied in the initial state defined by θ.

Rewriting logic has a computational reading of its inference rules that allows the specification of concurrent systems:[6] *reflexivity* means that a system may have *idle transitions*; *equality* means that *states* of a concurrent system are equal modulo the set of equations E; *congruence* is a general form of *sideways parallelism* in the sense that the arguments of the operator f may evolve in parallel; *nested replacement* combines an atomic transition at the top using a rule with nested concurrency in the substitution; *transitivity* is sequential composition.

It is important to discuss how such rewrite theory could be efficiently *executable* by some implementation. For this to happen, some requirements should be met [48]: the set of equations E should be decomposable into an union $E = E_0 \cup A$, with A a set of equational axioms such as associativity, commutativity, identity, for which an effective matching algorithm modulo A exists. Additionally, E_0 should be ground confluent and terminating (that is, applying the equations E_0 modulo A to a term t, we arrive in a finite number of rewrites into a single form). As for the rules R, they should be *coherent* [69] with E_0 modulo A, which means that, in order to rewrite in equivalence classes modulo E, we can always simplify a term with equations to its canonical form, and then rewrite with a rule in R. Finally, rules in R should be admissible ([13]), which intuitively means that there should be no free metavariables.

Finally, rewriting logic is reflective in the sense that its metatheory can be represented at the object level in a consistent way, so that the object-level correctly simulates the relevant metatheoretic aspects [37]. That is, there is a finite rewrite theory $\mathcal{U}$ that can simulate any other finitely representable rewrite theory $\mathcal{R}$ in the following sense: given any two terms t, t' in $\mathcal{R}$, there are corresponding terms $(\overline{\mathcal{R}}, \overline{t})$ and $(\overline{\mathcal{R}}, \overline{t'})$ in $\mathcal{U}$ such that we have:

$$\mathcal{R} \vdash t \to t' \quad \Leftrightarrow \quad \mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}}, \overline{t'})$$

[5] We follow here the rule as defined in [10], considering only conditional rewrites; in fact a general version of this rule also considers conditional equations and membership axioms.

[6] Based on [42].

With $\mathcal{U}$ being itself representable, giving rise to the so-called “reflective tower.”

$$\mathcal{R} \vdash \mathsf{t} \to \mathsf{t}' \quad \Leftrightarrow \quad \mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{\mathsf{t}}) \to (\overline{\mathcal{R}}, \overline{\mathsf{t}'}) \quad \Leftrightarrow \quad \mathcal{U} \vdash (\overline{\mathcal{U}}, \overline{(\overline{\mathcal{R}}, \overline{\mathsf{t}})}) \to (\overline{\mathcal{U}}, \overline{(\overline{\mathcal{R}}, \overline{\mathsf{t}'})}) \cdots$$

This characteristic of rewriting logic and its relation with the Maude interpreter is discussed on Section 2.3.2.

2.3 Maude

Maude names both the language and its implementation engine [14], a high-performance C++ implementation of rewriting logic that is capable of rewrites at the order of millions per second (see Appendix F for an example). As of Maude 2.1.1, only an interpreter is available, but a compiler is under development that promises to bring this number up to dozens of millions of rewrites per second. To keep this Section simple we opted to describe the aspects of Maude relevant to the implementation of *Maude MSOS Tool*. The complete description of its language is available in [14].

Maude implements rewriting logic theories and membership equational theories with *system modules* and *functional modules*, respectively. System modules are created with the keywords ‘`mod` n `is` D `endm`’ and functional modules are created with the keywords ‘`fmod` n `is` D `endfm`’, where n is the module name and D are the module declarations.

Module importation in Maude is made using one of the following keywords ‘`including`’ (or ‘`inc`’), ‘`extending`’ (‘`ex`’), and ‘`protecting`’ (‘`pr`’). The difference between the three types of inclusion is whether *junk* or *confusion* is allowed in the importation. Informally speaking, in algebraic specifications, *no confusion* is the requirement that different terms denote different things and *no junk* means that the algebra is *minimal*, since it has only the necessary elements. By including a module in ‘`protecting`’ mode, no junk and no confusion are allowed. For example, if we import ‘`BOOL`’ into a module ‘`FOO`’ in protecting mode, we are assuming that neither new constants of the sort ‘`Bool`’ will be created (no junk) nor any new meaning is added to the module ‘`BOOL`’, such as making the constants ‘`true`’ and ‘`false`’ equal (no confusion). The weaker form of inclusion ‘`extending`’ allows junk but does not allow confusion. (It is useful in our case when the data of an importing module is being extended with new constants, such as the signature of a programming language being divided in several modules.) The most general form of inclusion is by using the ‘`including`’ keyword, where junk and confusion may be introduced. Maude does not check whether to see if the requirements of the different forms of inclusion are being respected, since it would require theorem proving capabilities (but indeed some particular case could be verified).

The signature Σ in Maude is created by declaring sorts with the ‘`sort`’ keyword, subsorting relations with the ‘`subsort`’ keyword, operators with the ‘`op`’ keyword, and membership axioms with the ‘`mb`’ and ‘`cmb`’ keywords.

Let us exemplify these concepts by modeling *words* and *letters* in a language. We begin with the simple concept of letters, vowels, and consonants, represented by the sorts ‘`Letter`’, ‘`Vowel`’, ‘`Consonant`’, respectively.

```
sort Letter .
sort Vowel .
sort Consonant .
```

In order to give the order of the sorts on the poset $(S_k, \leq)$, one uses the '`subsort`' keyword. In this example, vowels and consonants are all letters, hence the subsorting relation that follows:

```
subsort Vowel < Letter .
subsort Consonant < Letter .
```

Let us add some operators to this signature. The operators that are part of the signature $\{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$ are defined using the '`op`' keyword with the following syntax:

```
op o : w̄ -> k [A] .
```

where o is the operator name, $\bar{w} = w_1 \cdots w_n$ are the domain sorts (or kinds), k the image sort (or kind), and A are the equational attributes: '`assoc`' defines associative operations, '`comm`' defines commutative operations, and '`id:t`' defines the term t as the identity of the operation being defined. Formally, as discussed on Section 2.2, this means that the rewrites and reductions will happen modulo these attributes. The '`ops`' keyword is a variant of '`op`' in which several operators may be defined at once, if they have the same domain and image sorts. In the following example, there is no domain sort—they are all *constants*. The '`constructor`' (or '`ctor`') attribute is not an equational attribute, but an indication that this operator is a *constructor* of terms. Constructors define the *structure* of the terms in the specification, while regular operators compute new terms from their arguments.

```
ops a e i o u : -> Vowel [constructor] .
ops b c d f g h j k l m n
      p q r s t v w x y z : -> Consonant [constructor] .
```

If we use the built-in command '`reduce`'—that essentially applies the equations in E to a Σ-term and is described in detail on Section 2.3.1—we may check that the vowel '`a`' is also a letter, as expected. The '`_::_`' operator is a built-in predicate in Maude: $t :: s$ checks if t has sort s.

```
Maude> reduce a :: Letter .
reduce in WORDS : a :: Letter .
result Bool: true
```

As expected, '`a`' is not a consonant, as the following '`reduce`' shows.

```
Maude> reduce a :: Consonant .
reduce in WORDS : a :: Consonant .
result Bool: false
```

Since the sorts 'Letter', 'Vowel', and 'Consonant' are all related, they form a single *connected component* and belong all to the same *kind*. This kind is not named explicitly, but one can use any sort name that is part of the connected component surrounded by brackets, such as '[Vowel]', as the kind name. By default Maude chooses the kind named by the topmost sort on the poset. In this case, it is '[Letter]'.

Let us add the concept of a *word* to our simple specification. We begin by defining a new sort 'Word'. To simplify this exposition, in this specification a single letter is a "trivial word."

```
sort Word .
subsort Letter < Word .
```

We now make use of Maude's capability of creating an operator with no name, usually called the "juxtaposition" operator, where the new term is constructed by putting each argument side by side.

```
op __ : [Word] [Word] -> [Word] [assoc] .
```

The use of the equational attribute 'assoc' means that this operator is associative. Intuitively this means that one may write 'a b c' instead of '(a b) c' and so on. We may now write something like 'c b v n' and it will be identified as a '[Word]'. The juxtaposition operator was declared over the kind '[Word]' with a purpose: recall that in MEL a term with a kind but not a sort is an error term. This fits well with our purpose, since we want *some* sequences of letters to be words, but not all.

We now add the capability of identifying actual words to our specification. We make use of Maude's 'mb' declaration to create a membership axiom that defines certain sequences of letters as words in our language. Its syntax is:

```
mb t : s .
```

where t is a term and s is a sort. For example:

```
mb  a b e t t e d    : Word .
mb  a b e t t e r    : Word .
mb  a b e t t i n g  : Word .
mb  a b e y a n c e  : Word .
mb  a b h o r        : Word .
           ...
```

Now, if we ask Maude to reduce 'a x q' we will see that it tells us that it is of the kind '[Word]' (no sort), while the reduction of 'a b h o r' gives the correct sort 'Word'. The topmost sort is now 'Word'.

```
reduce in WORDS : a x q .
result [Word]: a x q

reduce in WORDS : a b h o r .
result Word: a b h o r
```

We have thus exemplified the concept of *sorts*, *kinds*, and *operators* with equational attributes. We now exemplify the use of *equations* and *rewrite rules*. Let us assume the existence of a built-in module '`INT`' that defines the integers with sort '`Int`'. We begin by defining a *set of integers* ('`IntSet`'), which is represented by an associative-commutative operator. This set of integers has the identity '`null`'.

```
 sort IntSet .
 subsort Int < IntSet .

 op null : -> IntSet .
 op __ : IntSet IntSet -> IntSet [assoc comm id: null] .
```

As it was defined, the set of integers '`IntSet`' is actually a *multi-set*, since it allows the repetition of elements; let us add an equation that eliminates duplicated elements from an '`IntSet`'.

Metavariables in Maude specifications must be explicitly declared before they are used with the following syntax:

```
var ν : s .
```

where ν is the variable name and `s` its sort. An alternative form that avoids the predeclaration of metavariables is to use them explicitly in equations and rules using the syntax 'ν:s'.

Unconditional equations in Maude are written with the following syntax:

```
eq [L] : t = t′ .
```

where `t` and `t′` are terms that belong to the same kind and `[L]` is an optional label.

In the example below we declared the metavariable '`I`' to range over '`Int`'. Next we created an unconditional equation that specifies that any two repeated integers should be removed and a single copy should be kept. We could have written the equation as '`eq I:Int I:Int = I:Int .`', this way we would not have to declare the '`I`' metavariable.

```
var I : Int .
eq I I = I .
```

This single equation is sufficient, since Maude rewrites modulo equivalence classes—in this case, the associative-commutative equivalence class of the operator ('`__`'). Also, due to the *Congruence* deduction rule of MEL, the equation is applicable as many times as possible inside a term. For example a term such as '`1 2 1 2`' will match twice against '`I I`': the first match will be '`1 1`' and the second will be '`2 2`'.

```
reduce in INTEGER : 1 2 1 6 2 1 2 1 2 1 2 .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result IntSet: 1 2 6
```

The identity attribute of the '`__`' operator can be demonstrated with the following reduction example, where the identity '`null`' is, as expected, removed from the set of integers.

```
reduce in INTEGER : 1 2 3 null 4 3 1 null 1 3 .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result IntSet: 1 2 3 4
```

In order to demonstrate rewrite rules, let us create an operation that selects, non-deterministically, an integer out of an integer set. We begin by creating the operator:

```
op select : IntSet -> Int .
```

We now add the desired rule. In Maude, unconditional rules are written with the following syntax:

```
rl [L] : t => t′ .
```

where `t` and `t′` are terms that belong to the same kind, and `L` is an optional label.

The rule '`select`' below will non-deterministically select an integer out of an integer set due to the associative-commutative matching of the pattern '`I S`'.

```
rl [select] : select(I S) => I .
```

The '`rewrite`' command (also detailed in Section 2.3.1) attempts to rewrite a given term `t` until no further rewrite rules apply. In the example below it will only apply once to the term '`select(...)`', since it will rewrite this to an integer, to which no rewrite rule applies. (As a side note, the reason that the sort of '`1`' is '`NzNat`' is because Maude attempts to show the least sort applicable to a term in the output of the '`rewrite`' command. The module '`INT`' actually imports other modules (such as '`NAT`', the naturals) that defines a hierarchy of sorts involving '`NzNat`', the non-zero naturals, '`Nat`', the naturals, '`NzInt`', the non-zero integers, and '`Int`' itself. The least sort of '`1`' is '`NzNat`' in this case.)

```
rewrite in INTEGER : select(1 2 1 6 2 1 2 1 2 1 2) .
rewrites: 9 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 1
```

The chance of the rule selecting '1', '2', or '6' are exactly the *same*. This is because equations are applied *before* the rules are applied, due to the *coherence* requirement. This would reduce the term '1 2 1 6 2 1 2 1 2 1 2' to '1 2 6' and the rule would apply over this final term. In Section 2.3.1 we will see how built-in tools in the Maude interpreter can be used to *search* through all possible outcomes of non-deterministic specifications.

Now, let us conclude this Section with a word on conditions. We opted to make the examples so far very simple for ease of exposition, however, Maude also supports *conditional* membership axioms, equations, and rewrite rules, as we outlined formally in Section 2.2. They are specified using the following syntax:

For conditional membership axioms:

```
cmb t : s if C .
```

For conditional equations:

```
ceq t = t′ if C .
```

For conditional rewrite rules:

```
crl t => t′ if C .
```

The condition C is either a conjunction of conditions combined with the '_/_' operator, or one of the following:

- Ordinary equations 't = t'', which are satisfied if and only if the canonical forms of 't' and 't'' are equal modulo the equational attributes specified in the operators in 't' and 't'', such as associativity, commutativity, and identity.
- Abbreviated boolean equations such as 't', abbreviating the equation 't = true'. There are a number of built-in predicates, such as: equality (_==_), inequality (_=/=_), membership predicates (_:: S, with 'S' a sort, which returns true if the parameter is of sort 'S'), together with a combination of the connectives not_, _and_, and _or_.
- Matching equations [14], written as 't := t'', which are also ordinary equations, but with additional requirements at the operational level. In essence, matching equations are used to instantiate new variables by matching the left-hand side of the matching equation against the right-hand side.
- Rewrites, such as 't => t'', where 't' and 't'' are terms of any sort, which means that there is a rewrite of the term 't' to the term 't'' with zero or more rewriting steps.

Conditional rewrites must only be used in conditional rewrite rules, of course; otherwise all types of conditions may be used on membership axioms, equations, and rules.

2.3.1 Tools

We have already seen examples of some of the tools available on the Maude interpreter, namely the ability of reducing and rewriting terms. This Section will describe those commands in more detail, while also describing the *breadth first search* and *model checking* capabilities of Maude.

2.3.1.1 Reducing and rewriting terms

We begin by the '`reduce`' command, abbreviated with '`red`'. It receives an argument a term `t` and attempts to reduce this term to a normal form by selectively applying all applicable equations to it until no further equation is applicable.

The syntax is '`reduce t`', where `t` is a term to be reduced. Optionally, we may specify that the reduction should be made in module M by writing: '`reduce in M : t`'. For example:

```
Maude> red in NAT : 100 + 50 .
reduce in NAT : 50 + 100 .
rewrites: 1 in 0ms cpu (0ms real)
result NzNat: 150
```

A similar command is '`rewrite t`', abbreviated with '`rew`'. It attempts also to reduce a term `t` by first reducing it into a normal form `t'`, according to the equations, and then applying all applicable rewrite rules to it until no further rule is applicable. As opposed to equations, assumed Church-Rosser and terminating, rewrite rules may lead to non-termination; to cope with this, one may use an argument to this command that establishes an upper limit n on the number of rewrites performed, '`rewrite [n] t`'.

As an example, consider the following simple system module that increments the argument of the operator '`counter`' of sort '`Counter`' that takes as argument a natural (sort '`Nat`'). This module also shows an example of a conditional rewrite rule: the rule labelled '`inc`' is only applicable to '`counter(n)`' if n is less than 1000.

```
mod COUNTER is protecting NAT .
 sort Counter .
 op counter : Nat -> Counter .

 var n : Nat .

 crl [inc] : counter (n) => counter (n + 1)
 if n < 1000 .
endm
```

Issuing an unbounded '`rewrite`' command we arrive at the final possible value, '`counter(1000)`'.

```
Maude> rew counter(0) .
rewrite in COUNTER : counter(0) .
rewrites: 3001 in 10ms cpu (10ms real) (300100 rewrites/second)
result Counter: counter(1000)
```

However, if we use a *bounded* rewrite, we arrive at the upper bound given as parameter to the '`rew`' command, since, in this particular module, a single rewrite step increments by one the value of the counter.

```
Maude> rew [100] counter(0) .
rewrite [100] in COUNTER : counter(0) .
rewrites: 300 in 0ms cpu (0ms real) (~ rewrites/second)
result Counter: counter(100)
```

2.3.1.2 Searching for states

Another tool available is the breadth first search that is performed by the command '`search`'. Essentially, it attempts to find a *rewrite proof* from a term to a final pattern by applying the deduction rules of the rewriting calculus. The syntax of the command is (everything between { and } is optional):

```
search {[b]} {in m :} t R p {such that C}
```

where `b` is an upper bound on the number of solutions returned by the command, default is unbounded; `m` is the module in which the search will be made, default is the current module; `t` is the initial state in which the search will begin and `p` is the *pattern* of the final state; `R` is the *relation* between `t` and `p` and can be one of the following:

- '`=>1`': one step proof;
- '`=>+`': one or more steps proof;
- '`=>*`': zero or more steps proof;
- '`=>!`': only canonical final states are allowed.

An optional condition `C` may be specified to be satisfied by the rewrite proof.

To exemplify, let us return to our first example in which a number is selected non-deterministically from a set of integers.

```
mod INTEGER is protecting INT .
 sort IntSet .
 subsort Int < IntSet .

 op null : -> IntSet .
 op __ : IntSet IntSet -> IntSet [assoc comm id: null] .
```

```
 var S : IntSet .
 var I : Int .

 eq I I = I .

 op select : IntSet -> Int .

 rl select(I S) => I .
endm
```

If we search for all possible terms that are reachable beginning with the term '`select(1 2 1 6 2 1 2 1 2 1 2)`' we must use the pattern '`S:IntSet`'. If we want a proof that includes zero or more steps, we use the '`=>*`' relation. The pattern '`S:IntSet`' gets matched against each possible state reachable with zero or more steps beginning with '`select(1 2 1 6 2 1 2 1 2 1 2)`'. Also recall that rewrite rules in Maude are *coherent* and all terms are reduced to a normal form before the rewrite rules are applied, hence the term '`select(1 2 1 6 2 1 2 1 2 1 2)`' is first reduced to '`select(1 2 6)`' before the search begins.

```
search in INTEGER : select(1 2 1 6 2 1 2 1 2 1 2) =>* S .

Solution 1 (state 0)
states: 1  rewrites: 8 in 0ms cpu (0ms real)
S --> select(1 2 6)

Solution 2 (state 1)
states: 2  rewrites: 9 in 0ms cpu (0ms real)
S --> 1

Solution 3 (state 2)
states: 3  rewrites: 10 in 0ms cpu (0ms real)
S --> 2

Solution 4 (state 3)
states: 4  rewrites: 11 in 0ms cpu (0ms real)
S --> 6

No more solutions.
states: 4  rewrites: 11 in 0ms cpu (10ms real)
```

If we want only final states, that is, states in which no more rewrite rules are applicable, we use the '`=>!`' relation:

```
search in INTEGER : select(1 2 1 6 2 1 2 1 2 1 2) =>! S .

Solution 1 (state 1)
```

```
states: 4  rewrites: 11 in 0ms cpu (0ms real)
S --> 1

Solution 2 (state 2)
states: 4  rewrites: 11 in 0ms cpu (0ms real)
S --> 2

Solution 3 (state 3)
states: 4  rewrites: 11 in 0ms cpu (0ms real)
S --> 6

No more solutions.
states: 4  rewrites: 11 in 0ms cpu (0ms real)
```

2.3.1.3 Model checking specifications

Maude comes with a model checker that supports *linear temporal logic* (LTL) formulae. This Section gives a brief overview of its model checking capabilities and how LTL formulae are encoded in Maude. The information present in this Section is based on [14, Chapter 9].

Let us describe inductively the set of formulae of the propositional linear temporal logic LTL(AP) over a set AP of atomic propositions. We also give their concrete signature defined by the module '`MODEL-CHECKER`'.

- $\top \in$ LTL(AP) always satisfiable, written as '`True`';
- if $\varphi \in$ LTL(AP), then $\bigcirc\varphi \in$ LTL(AP) is the *next operator*, which holds if φ is satisfiable at the next step of computation, written as '`O` φ';
- if $\varphi, \psi \in$ LTL(AP), then $\varphi\,\mathcal{U}\,\psi \in$ LTL(AP) is the *strong until operator*, which holds if, during the computation, φ is valid, until ψ becomes valid, written as 'φ `U` ψ';
- *atomic propositions*, if $\mathsf{p} \in \mathsf{AP}$ then $\mathsf{p} \in$ LTL(AP), and are defined by operators in Maude whose image sort is '`Prop`';
- *boolean connectives*, if $\varphi, \psi \in$ LTL(AP) then $\neg\varphi$ and $\varphi \vee \psi$ are in LTL(AP), written as '`~_`' and '`_or_`'.

Some useful syntactic sugar may be defined over this minimal set of formulae, as Table 2.1 shows.

Kripke structures are the natural models for propositional temporal logic. Essentially, a Kripke structure is a total unlabeled transition system to which a set of unary state predicates have been added on its set of states. Formally, it is a triple $\mathcal{A} = (\mathsf{A}, \rightarrow_{\mathcal{A}}, \mathsf{L})$, where A is a set of states, $\rightarrow_{\mathcal{A}}$ is a total binary relation on A, called the transition relation, and $\mathsf{L} : \mathsf{A} \rightarrow \mathcal{P}(\mathsf{AP})$ is a function, called the labeling function, associating to each state $\mathsf{a} \in \mathsf{A}$ the set $\mathsf{L}(\mathsf{a})$ of those atomic propositions in AP that hold in the state a. In a rewriting logic system module that specifies a rewrite theory $\mathcal{R} = (\Omega, \mathsf{E}, \mathsf{R})$, the Krikpe

Name	*Formula*	*Equivalent formula*	*Maude formula*
false	$\bot$	$\neg\top$	`False`
conjunction	$\varphi \wedge \psi$	$\neg((\neg\varphi) \vee (\neg\psi))$	`φ /\ ψ`
implication	$\varphi \rightarrow \psi$	$(\neg\varphi) \vee \psi$	`φ |-> ψ`
eventually	$\Diamond\varphi$	$\top\,\mathcal{U}\,\varphi$	`<>φ`
henceforth	$\Box\varphi$	$\neg\Diamond\neg\varphi$	`[]φ`
release	$\varphi\,\mathcal{R}\,\psi$	$\neg((\neg\varphi)\,\mathcal{U}\,(\neg\psi))$	`φ R ψ`
unless	$\varphi\,\mathcal{W}\,\psi$	$(\varphi\,\mathcal{U}\,\psi) \vee (\Box\varphi)$	`φ W ψ`
leads-to	$\varphi \rightsquigarrow \psi$	$\Box(\varphi \rightarrow (\Diamond\psi))$	`φ |-> ψ`
strong implication	$\varphi \Rightarrow \psi$	$\Box(\varphi \rightarrow \psi)$	`φ => ψ`
strong equivalence	$\varphi \Leftrightarrow \psi$	$\Box(\varphi \leftrightarrow \psi)$	`φ <=> ψ`

Table 2.1: LTL formulae derived from the minimal set

structure has as a set of states A the set $T_{\Omega/E,k}$ which is the set of *canonical terms* of the kind k, the transition relation $\rightarrow_{\mathcal{A}}$ is the *one-step* rewriting transitions of terms of kind k and the labeling function $L(a)$ is the set of atomic propositions defined equationally over terms of kind k that hold for state $a \in A$.

Let us give a very simple example, a simple state machine with elements '`a`', '`b`', '`c`', '`d`', and '`f`', of sort '`Elt`'. States are created with the operator '`st_`' that receives as single argument an '`Elt`'. The transitions of this state machine are given by the labelled rewrite rules in the module '`STATE-MACHINE`' below. The two equations are there only to demonstrate that the state space is indeed formed by canonical terms generated by the specification, that is, '`a`'–'`f`'. (Section 6.3 and Appendix E contain several examples of the use of Maude's model checker for a variety of specifications.)

```
mod STATE-MACHINE is
 sorts St Elt .

 op st_ : Elt -> St .
 ops a b c d e f BB CC : -> Elt .

 eq BB = b .
 eq CC = c .

 rl [a->b] : st a => st BB .
 rl [a->c] : st a => st CC .
 rl [b->a] : st b => st a .
 rl [c->d] : st c => st d .
 rl [c->e] : st c => st e .
 rl [d->f] : st d => st f .
 rl [f->a] : st f => st a .
endm
```

In order to use the model checker, we first need to define what is the *state space* of our specification. In this case it is clearly the space defined by the sort '`St`' with the operator

'`st_`'. In module '`CHECK-STATE-MACHINE`' below, after including the '`MODEL-CHECKER`' module, we create the subsort relation between '`St`' and '`State`', which is a built-in sort in the '`MODEL-CHECKER`' module that represents the state space that will be explored by the model checking algorithm.

```
mod CHECK-STATE-MACHINE is
 protecting STATE-MACHINE .
 including MODEL-CHECKER .

 subsort St < State .
 ...
endm
```

We are now ready to implement a simple proposition to be checked. It must be an operator whose image sort is '`Prop`'. We must define when this predicate holds, and it is done by creating an equation involving the following operator:

$$\texttt{_:=_} : \texttt{State} \times \texttt{Formula} \rightharpoonup \texttt{Bool}$$

where '`Formula`' is either a proposition, one of the formulae present in LTL(AP), described at the beginning of this Section, or one of the formulae shown in Table 2.1.

In the example below we created a proposition '`in s`' that will hold only when the current state is s. Since the operator '`_:=_`' is partial, there is no need to specify when a predicate is false.

```
 var n : Elt .
 op in_ : Elt -> Prop .
 eq st n |= in n = true .
```

Figure 2.2 shows the Kripke structure associated with the '`STATE-MACHINE`' system module. The states are the canonical terms '`a`'–'`f`', the arrows represent the one-step transitions between each state. Although we did not make it explicit in the figure, each state s has an associated propositional formula '`in s`'. For example, in state a the proposition '`in a`' is true, while '`in b`', '`in c`', '`in d`', and '`in f`' are false.

In order to verify a LTL formula, we need to use the following operator:

$$\texttt{modelCheck} : \texttt{State} \times \texttt{Formula} \rightharpoonup \texttt{ModelCheckResult}$$

where the first parameter is the initial state, and the second the LTL formula to be verified. The image '`ModelCheckResult`' is either '`true`' or a counterexample to the LTL formula.

As an example of model checking an LTL formula, let us verify that, given the specification '`STATE-MACHINE`' above, if state '`b`' *never occurs* ('`~ <> in b`'), then eventually the current state will be '`f`' ('`<> in f`').

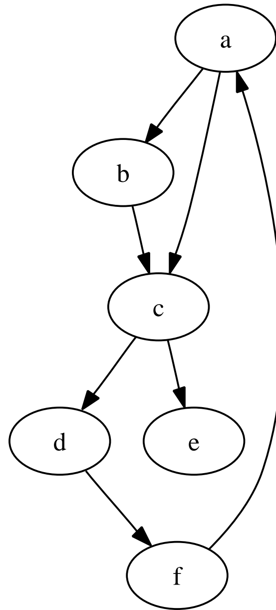

Figure 2.2: (Simplified) Kripke structure for ‘`STATE-MACHINE`’

```
reduce in CHECK-STATE-MACHINE :
 modelCheck(st a, ~ <> in b -> <> in f) .
rewrites: 21 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult:
 counterexample({st a,'a->c} {st c,'c->e}, {st e, deadlock})
```

The verification fails, giving us a counterexample, a pair of lists of transitions ‘`counterexample(t,` where `t` corresponds to a finite path beginning at the initial state (‘`st a`’), and `t'` describes a loop (state ‘`st e`’ is a dead end and can only rewrite to itself through the *Reflexivity* rule of rewriting logic, see Section 2.2). The counterexamples in the model checker have this form because if an LTL formula φ is not satisfied by a finite Kripke structure, it is always possible to find a counterexample for φ having the form of a path of transitions followed by a cycle (see [14, Chapter 9]).

2.3.2 Metalevel programming

The universal and reflective aspects of rewriting logic discussed at the end of Section 2.2 is realized as the ‘`META-LEVEL`’ module in Maude's prelude. This module defines a metarepresentation of all Maude constructions, such as sorts, constants, variables, terms, modules, equations, rules, etc. Let us describe some of the most relevant functions in this module that will be important in the implementation of the *Maude MSOS Tool*.

The so-called “up” functions convert terms in their object form to the metarepresentation. The function ‘`upModule(q)`’ returns the metarepresentation, a term of sort ‘`Module`’, of the module named by its first argument `q`, a quoted-identifier such as ‘`'NAT`’. The second argument instructs ‘`upModule`’ to return (or not) a *flattened* metamodule: ‘`true`’ means to include all dependent modules on the metarepresentation. The function is partial, since the given module may not exist in Maude's internal database of loaded modules.

$$\texttt{upModule} : \texttt{Qid} \times \texttt{Bool} \rightharpoonup \texttt{Module}$$

As an example let us obtain by metarepresentation of the internal module 'TRUTH-VALUE'. The omitted contents of the 'special' attributes is of no importance here and are related to the actual implementation of the Maude engine.

```
Maude> red in META-LEVEL : upModule('TRUTH-VALUE, false) .
reduce in META-LEVEL : upModule('TRUTH-VALUE, false) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result FModule: fmod 'TRUTH-VALUE is
  nil
  sorts 'Bool .
  none
  op 'false : nil -> 'Bool [ctor special(...)] .
  op 'true : nil -> 'Bool [ctor special(...)] .
  none
  none
endfm
```

The function 'upTerm(u)' converts a term u from its object form (represented here by a term of the built-in sort 'Universal') into its metarepresentation.

$$\texttt{upTerm} : \texttt{Universal} \rightarrow \texttt{Term}$$

For example, 'upTerm(true)' gives the metarepresentation of the constant 'true'.

```
Maude> red in META-LEVEL : upTerm(true) .
reduce in META-LEVEL : upTerm(true) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Constant: 'true.Bool
```

The example of the 'TRUTH-VALUE' module above shows that the metarepresentation of functional and system modules in Maude is very close to the object level. Let us also briefly discuss the notation for metaterms. *Constants* are represented by a quoted-identifier that consists of the constant name, a period, and the sort name. For example, as the example at the beginning of this paragraph shows, the metarepresentation of the constant 'true' of the sort 'Bool' is ''true.Bool'. The metarepresentation of *metavariables* follows the same pattern, but uses a colon to separate the metavariable name and its sort, such as: ''b:Bool'. Non-constant operators are represented by the operator name, including the underscores used in the mixfix notation, with the arguments surrounded by brackets. For example, in a module that defines the sort 'Foo', with constants 'a' and 'b', and a binary mixfix operation '_._', the metarepresentation of 'a . b' is ''_._['a.Foo,'b.Foo]'.

Functions that move from the metalevel to the object level are the "down" functions. As of Maude 2.1.1, there is no counterpart of 'upModule', that is, a function that moves from a metamodule to a module. The opposite of 'upTerm' is the function that moves

from a metarepresentation into an object form, ‘`downTerm`’. It receives as first argument the metaterm that will be converted into the term. The second argument is a term that will act as an “error term” in case the function does not succeed in the conversion.

$$\texttt{downTerm} : \texttt{Term} \times \texttt{Universal} \rightarrow \texttt{Universal}$$

As an example, let us convert back from ‘`'true.Bool`’ into an object form by calling ‘`downTerm('true.Bool, error-bool)`’. Here, ‘`error-bool`’ is a previously created constant that will be returned as value if the conversion is unsuccessful. The result is, as expected, ‘`true`’, a term of sort ‘`Bool`’. If we try to convert a bogus term, such as ‘`'a.Bool`’, Maude will generate a warning and the resulting term will be ‘`error-bool`’, as follows:

```
reduce in BOOL-META : downTerm('a.Bool, error-bool) .
Advisory: could not find a constant a of sort Bool in
          meta-module BOOL-TEST.
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: error-bool
```

Finally, let us finalize this description with four functions that further enhance Maude’s metaprogramming capabilities. The function ‘`metaParse`’ constructs a metaterm out of a sequence of quoted-identifiers (the second argument), using the signature defined by the first argument, a metamodule. The expected type to be parsed is given as third argument—or ‘`anyType`’, if the type is not known *a priori*. The function is partial and results in a term of sort ‘`ResultPair?`’ that contains either a tuple with the metaterm and its type or an error message.

$$\texttt{metaParse} : \texttt{Module} \times \texttt{QidList} \times \texttt{Type?} \rightharpoonup \texttt{ResultPair?}$$

The sequence of quoted-identifiers is used because as we shall see in Section 2.3.4, the ‘`LOOP-MODE`’ facility in Maude converts all user input into a sequence of quoted-identifiers. If the user enters, say, ‘`(mod F is sort A . endm)`’, the ‘`LOOP-MODE`’ converts this into ‘`'mod 'F 'is 'sort 'A '. 'endm`’.

As an example, let us use the signature defined by the built-in module ‘`BOOL`’ to parse the string ‘`true and false`’. We must pass this string to the ‘`metaParse`’ function as ‘`'true 'and 'false`’. The result is a ‘`ResultPair`’ containing the parsed term and its type, as follows. We used the function ‘`upModule('BOOL, false)`’ to obtain the metarepresentation of the built-in module ‘`BOOL`’.

```
reduce in META-LEVEL : metaParse(upModule('BOOL, false),
    'true 'and 'false, 'Bool) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'_and_['true.Bool,'false.Bool],'Bool}
```

Giving a bogus input to the ‘`metaParse`’, the result is ‘`noParse(n)`’ where `n` is the position of the problematic qid. In the example below, the function could not parse the third qid.

```
reduce in META-LEVEL : metaParse(upModule('BOOL, false),
    'true 'and '3, 'Bool) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair?: noParse(2)
```

The counterpart of 'metaParse' is 'metaPrettyPrint', whose function is to take a signature, a term, and to return its quoted-identifier representation.

$$\texttt{metaPrettyPrint} : \texttt{Module} \times \texttt{Term} \to \texttt{QidList}$$

By passing the term ''_and_['true.Bool,'false.Bool]' to this function, along with the signature provided by the functional module 'BOOL', we obtain back the qid list ''true 'and 'false'.

```
reduce in META-LEVEL : metaPrettyPrint(upModule('BOOL, false),
    '_and_['true.Bool,'false.Bool]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result TypeList: 'true 'and 'false
```

(The reason why Maude gives the sort of ''true 'and 'false' as 'TypeList' is because this sort is a subsort of 'QidList', and, as we mentioned on Section 2.3, Maude attempts to print the least sort applicable to a term.)

In order to effectively execute rewriting logic modules at the metalevel, we need functions that reduce and rewrite metaterms. The first is 'metaReduce', which takes as input a metamodule, a metaterm and returns a 'ResultPair' containing the term and its type resulting from metareducing the metaterm against the metamodule.

$$\texttt{metaReduce} : \texttt{Module} \times \texttt{Term} \rightharpoonup \texttt{ResultPair}$$

The 'metaRewrite' is the equivalent of the 'rewrite' command and, like 'metaReduce', receives as arguments a metamodule, a metaterm, and an upper bound on the number of rewrites. The result is also a 'ResultPair' with a term and its sort.

$$\texttt{metaRewrite} : \texttt{Module} \times \texttt{Term} \times \texttt{Bound} \rightharpoonup \texttt{ResultPair}$$

Let us give an example of the application of the 'metaReduce' function. Consider the following metamodule, 'FOO', that contains two constants and a function 'f', whose value is defined by the single equation present on the module.

```
fmod 'FOO is
  protecting 'BOOL .
  sorts 'Foo .
  none
  op 'a : nil -> 'Foo [none] .
  op 'b : nil -> 'Foo [none] .
```

```
  op 'f : 'Foo -> 'Foo [none] .
  none
  eq 'f['a.Foo] = 'b.Foo [none] .
endfm
```

Let us execute an instance of the '`metaReduce`' function as follows:

```
reduce in META-LEVEL :
 metaReduce(fmod 'FOO ... endfm, 'f['a.Foo]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'b.Foo,'Foo}
```

The result of the '`metaReduce`' operation is a '`ResultPair`' with the metarepresentation of the constant '`b`' along with its sort, '`Foo`'.

The functionality of '`metaRewrite`' is similar. We exemplify its use by creating the metarepresentation of the the '`COUNTER`' module, used as an example on Section 2.3.1. The numbers are in Peano notation, with a constant '`0`' and the successor function '`s_`'. The notation '`s_^1000 (0)`', metarepresented by '`s_^1000['0.Zero]`' uses Maude's *iterated operator* syntax, which n applications of an operator `f` to an argument x may be written as '`f^`n `(x)`' if the operator `f` was defined with the '`iter`' attribute.

```
result SModule: mod 'COUNTER is
  protecting 'BOOL .
  protecting 'NAT .
  sorts 'Counter .
  none
  op 'counter : 'Nat -> 'Counter [none] .
  none
  none
  crl 'counter['n:Nat] => 'counter['_+_['n:Nat,'s_['0.Zero]]]
  if '_<_['n:Nat, 's_^1000['0.Zero]] = 'true.Bool [label('inc)] .
endm
```

Let us execute an instance of the '`metaRewrite`' function, with an upper bound on the limit of rewrites of 20. The result is the metarepresentation of the term '`counter(20)`', with sort '`Counter`', as expected.

```
reduce in META-LEVEL : metaRewrite(mod 'COUNTER ... endm,
                                   'counter['0.Zero], 20) .
rewrites: 61 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'counter['s_^20['0.Zero]],'Counter}
```

2.3.2.1 Maude as meta-tool

The reflective capabilities of the Maude system enables the possibility of using it as a *formal meta-tool* [15, 12]. In order to be formal, a tool must support a *precise axiomatization*

of the language it is implementing. This is different from writing tools in conventional languages such as C, or Java, since the implementation is not a formal axiomatization.

The point is that, as mentioned in Section 2.2, RWL is a *logical framework* which can represent in a *natural* way many different logics, languages, operational formalisms and models of computation. This natural representation is the result of using MEL together with equational attributes such as associativity, commutativity, providing a very general representation framework with a simple calculus. The formal aspect regards, of course, all the logical aspects of RWL and MEL discussed in Section 2.2, together with their executability in Maude.

The representation of a logic $\mathcal{L}$ in rewriting logic is given by a representation map:

$$\Psi : \mathcal{L} \rightarrow \mathcal{R}$$

Combining the flexible syntax given by its equational logic together with the reflective capabilities outlined in this Section this map can be implemented as an executable function $\overline{\Phi}$ with the following signature:

$$\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \rightarrow \texttt{Module}$$

in a module that extends '`META-LEVEL`'. Here, $\texttt{Module}_{\mathcal{L}}$ is an abstract data type that represents theories in the logic $\mathcal{L}$. By using the descent functions such as '`metaReduce`', '`metaRewrite`', it is possible to *execute* $\mathcal{L}$ in Maude.

2.3.3 Input and output facilities

Input and Output in Maude is made using the '`LOOP-MODE`' facility. This module provides a built-in mechanism for accepting user input and printing text back to the user. The basic construction of the '`LOOP-MODE`' is the *loop object*, of sort '`System`', whose signature is as follows:

```
mod LOOP-MODE is
 protecting QID-LIST .

 sorts State System .

 op [_,_,_] : QidList State QidList ->
              System [ctor special (...)] .
endm
```

The loop object, '`[_,_,_]`' consists of three arguments: the first is the *input stream*, which is the representation of the user input; the second is an abstract term of sort '`State`' that is to be concretely defined by the user and is application dependent; the third is the *output stream*.

This object functions in a read-eval-print loop as follows. First, one creates an initial loop object by using the '`[_,_,_]`' operator and passes this object to the '`loop`' command.

Next the user must input all text between parentheses. Everything sent this way will be converted into a sequence of *tokens* (which is another name for quoted-identifiers). From this point on, the system must have rules that make all possible types of user input and process it (possibly modifying the '`State`' of the loop object), and then prints the result of the operation by putting a sequence of tokens back on the third argument of the loop object.

Let us exemplify this by creating a simple read-eval-print loop that accepts as input a simple arithmetic expression and prints back its value. We begin by creating an initial loop object that greets the user and asks for input. The state of this read-eval-print loop records the last value computed.

```
op last-value : Int -> State .

op init : -> System .
eq init = [ nil, last-value (0),
            'Hello! 'Please 'type 'an 'expression. ] .
```

We begin with a simple rule: typing '`(*)`' at the prompt will print the last computed value. On the rule below '`QIL`' will match against any token list that is going to be printed on the output stream and it appends the message to this list. The built-in function '`string`' converts a number to a string according to the given base (10, in this case). The function '`qid`' converts a string into a quoted-identifier to be put on the output stream.

```
rl [ '*, last-value (n), QIL ] =>
   [ nil, last-value (n), QIL
     'Last 'computed 'value: qid(string(n,10)) ] .
```

Let us define the signature of our simple language for arithmetic expressions, together with an operator used to calculate its final value. The module '`SIMPLE-LANGUAGE`' below defines two sorts, '`Op`' and '`Exp`', that represent, respectively, the arithmetic operations and expressions in this language. Expressions are formed by the operator '`___`' that receives two expressions and an operator. We also make our expressions range over the integers by subsorting '`Int`' to '`Exp`'.

```
fmod SIMPLE-LANGUAGE is
 protecting INT .

 sort Op .
 ops plus minus times div : -> Op .
 sort Exp .
 subsort Int < Exp .
 op ___ : Exp Op Exp -> Exp .
```

The operator '`[[_]]`' below receives as input an expression and returns its value by recursively calculating the values of each expression of the '`___`' operator. The value of an integer is the end of the recursion.

```
 vars E1 E2 : Exp .
 var  I : Int .

 op [[_]] : Exp -> Int .
 eq [[E1 times E2]] = [[E1]] * [[E2]] .
 eq [[E1 plus  E2]] = [[E1]] + [[E2]] .
 eq [[E1 minus E2]] = [[E1]] - [[E2]] .
 eq [[E1 div   E2]] = [[E1]] quo [[E2]] .
 eq [[I]] = I .
endfm
```

For the rules that actually receive an expression and calculate its value, we need to parse the user input using the signature from the 'SIMPLE-LANGUAGE' module. If the parsing is successful the rule does the following: the parsing of the user input ('QIL') will generate a metaterm 't'; this metaterm 't' is converted back into object form (an expression) by 'downTerm' and bound to the metavariable 'exp'; we use the '[[_]]' operator to obtain the value of 'exp' and bind it to the metavariable 'n''; we then use 'metaPrettyPrint' to convert (the metarepresentation of) 'n'' into the sequence of tokens that will be printed to the user. The rule also updates the 'last-value' state with this value.

```
 crl [ QIL, last-value(n), QIL' ] =>
     [ nil, last-value (n'), QIL'
       'Result: metaPrettyPrint (SIMPLE-LANGUAGE, upTerm(n')) ]
 if QIL =/= nil /\
    t := getTerm (metaParse (SIMPLE-LANGUAGE, QIL, 'Exp)) /\
    exp := downTerm (t, error-exp) /\
    n' := [[exp]] .
```

If we are unable to parse the user input, we must generate a "syntax error." The following rule will only be applied when the result of 'metaParse' is not of the sort 'ResultPair', which means that the parsing has failed:

```
 crl [ QIL, last-value(n), QIL' ] =>
     [ nil, last-value (n), QIL' 'Syntax 'error! ]
 if QIL =/= nil /\
    not (metaParse (SIMPLE-LANGUAGE, QIL, 'Exp) :: ResultPair) .
```

We may now show a transcription of a complete session using our little interpreter.

```
Hello! Please type an expression.
Maude> (10 plus 30 minus 30)
Result: 10
Maude> (hello)
Syntax error!
Maude> (20 plus 30)
```

```
Result: 50
Maude> (50 times 200)
Result: 10000
Maude> (*)
Last computed value: 10000
```

2.3.4 Full Maude

Full Maude is a Maude application that makes extensive use of the reflective power of rewriting logic and Maude. Full Maude defines a rich module algebra which includes module hierarchies, parameterization, views, theories, modules expressions, and object-oriented modules. It builds upon the capabilities described on Sections 2.3.2 and 2.3.3 to implement this functionality.

This Section describes some of the features of Full Maude that are used on the conversion process detailed in Chapter 5. Full Maude uses the '`LOOP-MODE`' facility, so all user input must be made between parentheses.

2.3.4.1 System and functional modules

Full Maude implements standard system and functional modules. All modules given to the Full Maude interpreter are stored in an internal *database of modules*. It also supports the standard Maude commands, such as '`reduce`', '`rewrite`', and '`search`'. This is possible because these commands are available at the meta-level as '`metaReduce`', '`metaRewrite`', and '`metaSearch`'.

As an introductory example, let us show a complete Maude session, starting by the loading of the Full Maude interpreter, inputting some module '`FOO`', and a rewrite example:

```
                    \||||||||||||||||||/
                  --- Welcome to Maude ---
                    /||||||||||||||||||\
            Maude 2.1.1 built: Jun 15 2004 12:55:31
             Copyright 1997-2004 SRI International
                   Mon Apr 11 11:35:13 2005
Maude> load full-maude

               Full Maude 2.1.1 (July 20th, 2004)

Maude> (mod FOO is
         sort Foo .
         ops a b : -> Foo .
         rl a => b .
       endm)

rewrites: 717 in 119ms cpu (119ms real) (5975 rewrites/second)
Introduced module FOO
```

```
Maude> (rew a .)
rewrites: 123 in 6ms cpu (6ms real) (17576 rewrites/second)
rewrite in FOO :
  a
result Foo :
  b
```

2.3.4.2 Theories, views, and parameterized modules.

Full Maude implements parameterized modules and views that enables a *parameterized style* of programming, in the OBJ [23] fashion. Parameterized programming in Full Maude combines three elements: the *parameterized modules* themselves that are defined over a set of parameters, *theories* define the structure and properties required of an actual parameter, and *views* map the formal interface theory to the corresponding actual module. In other words, views provide the interpretation of the actual parameters. To simplify the explanation we will avoid defining these concepts formally. See [23, 25] for more information.

Theories are used to declare module interfaces and are created with the syntax '`fth` n `is` D `endfth`' for functional theories, and '`th` n `is` D `endth`' for system theories, where n is the theory name and D its declarations. The most basic theory is '`TRIV`', predefined by Full Maude, which has a single sort, '`Elt`', as a requirement.

```
fth TRIV is
 sort Elt .
endfth
```

More complex theories can be created such as the following theory of monoids, which requires a constant '`1`' and a monoid binary operator, with identity '`1`'.

```
fth MONOID is including TRIV .
 op 1 : -> Elt .
 op __ : Elt Elt -> Elt [assoc id: 1]
endth
```

We only require the use of '`TRIV`' in the implementation of the *Maude MSOS Tool*, so we will refrain from discussing these more complex theories and their implication for parameterized programming. We invite the interested reader to see [14, Chapter 15] for more details.

As for views, we will discuss the more trivial mapping, since this is the one needed by the implementation of *Maude MSOS Tool*. Essentially, we need a view that maps the requirement of the sort '`Elt`' to a concrete sort, defined in some other module. The syntax of this type of view is:

```
view ν from TRIV to M is
```

```
 sort Elt to s .
endv
```

where ν is the view name, and s is a sort defined in M. It is common practice to make the view name the same as the sort name. For example, the following view maps the sort 'Elt' to the sort 'Exp', defined in the module 'EXP'.

```
view Exp from TRIV to EXP is
 sort Elt to Exp .
endv
```

Parameterized modules can be parameterized by one or more theories. In the case of the implementation of *Maude MSOS Tool*, only the 'TRIV' theory is used as a parameter. One declares a parameterized module with the following syntax for the module name: '$M(X_1::T_1 \mid \cdots \mid X_n::T_n)$', where $X_1, \ldots, X_n$ are the labels and $T_1, \ldots, T_n$ are the parameter theories. In Full Maude 2.1.1, all the sorts coming from theories in the interface must be qualified by their labels. If Z is the label of a parameter theory T, then each sort s in T has to be qualified as '$Z@s$'. As a concrete example, let us consider a parameterized module that defines sets:

```
fmod SET(X :: TRIV) is
 sorts Set(X) .
 subsort X@Elt < Set(X) .

 op null : -> Set(X) .
 op __ : Set(X) Set(X) -> Set(X) [assoc comm id: null] .

 var E : X@Elt .

 eq E E = E .
endfm
```

The name of the module 'SET(X :: TRIV)' indicates that it has a single parameter labelled 'X', with associated theory 'TRIV'; the sort 'Set(X)' is a *parameterized sort* name that uses as parameter the label 'X'. In general, given a parameterized module '$M(X_1::T_1 \mid \cdots \mid X_n::T_n)$', any sort s can be written in the form '$s(X_1 \mid \cdots \mid X_n)$'. Finally, as we mentioned, we used 'X@Elt' to refer to the sort coming from the 'TRIV' theory.

The actual instantiation of a parameterized module is through the importation of a specific *module expression* with the following syntax: '$M(\nu_1 \mid \cdots \mid \nu_n)$', where M is a parameterized module name and $\nu_1, \ldots, \nu_n$ are the views that maps the theories to sorts. For example, by defining a view from 'Elt' to 'Int' such as:

```
view Int from TRIV to INT is
 sort Elt to Int .
endv
```

we may use this view to create the module expression '`SET(Int)`' that has the effect of importing a module that defines the set of integers.

```
fmod TEST is
 including SET(Int) .
endfm
```

After importing all these modules, and views into Full Maude, we may issue a '`reduce`' command such as:

```
Maude> (red 1 2 3 4 .)
reduce in TEST :
  1 2 3 4
result Set‘(Int‘) :
  1 2 3 4
```

2.3.4.3 Extending Full Maude

User input in Full Maude is handled by the module '`FULL-MAUDE`', which contains the main input loop, using Maude's built-in '`LOOP-MODE`' facility. This module contains a series of rewrite rules that matches against specific Full Maude commands, and calls the appropriate functions to handle that input. Recall from Section 2.3.4 that user input that is entered between parentheses is converted by '`LOOP-MODE`' into a sequence of tokens. By passing this sequence of tokens along with the appropriate signature to the '`metaParse`' function, we obtain a metarepresentation of the input. Initially, Full Maude parses all user input using its own signature module, named '`GRAMMAR`'.

Once this metaparsing is completed, Full Maude must call the appropriate function to handle the command/module given by the user. For that to work the '`FULL-MAUDE`' module includes a number of modules, among them '`DATABASE-HANDLING`'. It is this module that handles user input after it is parsed by '`metaParse`' by calling the appropriate function. Which function to call is dependent to the term parsed by '`metaParse`'.

The entire cycle of reading user input and executing it is managed through an *object* of type '`Database`'. This object contains several attributes, including the input tokens, output tokens, the current module name, and the database of modules, which contains the metarepresentation of all modules that have been loaded by the user. This object has the following signature.

```
sort DatabaseClass .
subsort DatabaseClass < Cid .
op db :_ : Database -> Attribute .
op input :_ : TermList -> Attribute .
op output :_ : QidList -> Attribute .
op default :_ : ModName -> Attribute .
```

The module 'DATABASE-HANDLING' defines a class named 'DatabaseClass', subsort of 'Cid', which is the built-in sort that represents "class identifiers." The 'db' attribute holds the database of modules, which is the main construction of Full Maude. It contains the metarepresentation of all user inputted modules. The 'input' attribute holds the user input after the metaparsing is completed. The 'output' attribute is a 'QidList'—anything that is put into this attribute will be displayed to the user. Finally, the attribute 'default' holds the current module being used. This is necessary so that Full Maude knows which module to use when a 'reduce', 'rewrite', or 'search' command is called.

In order to extend Full Maude to add support for a language $\mathcal{L}$ one needs to create a *signature* for this language using a functional module. This module (say $\mathtt{GRAMMAR}_{\mathcal{L}}$) defines a data type $\mathrm{D}_{\mathcal{L}}$ that represents user input of programs in the language $\mathcal{L}$. It is usually desirable that this extension to be *conservative* one, that is, the "original" Full Maude language is still available, with the same semantics as usual. This is done by combining both signatures, GRAMMAR and $\mathtt{GRAMMAR}_{\mathcal{L}}$ into a single signature, say, GRAMMAR+FM+L. The rules for parsing user input must be changed to handle this new signature. For example the following rule receives user input and puts it into the 'input' attribute of the object.

```
crl [in] :
  [QIL,
   < O : X@Database |
       db : DB, input : nilTermList, output : nil,
       default : MN, Atts >,
   QIL']
  => [nil,
      < O : X@Database | db : DB,
          input : getTerm(metaParse(GRAMMAR, QIL, 'Input)),
          output : nil, default : MN, Atts >,
       QIL']
  if QIL =/= nil /\
     metaParse(GRAMMAR, QIL, 'Input) : ResultPair .
```

Rule '[in]' must be changed to:

```
crl [in] :
  [QIL,
   < O : X@Database |
       db : DB, input : nilTermList, output : nil,
       default : MN, Atts >,
   QIL']
  => [nil,
      < O : X@Database | db : DB,
         input : getTerm(metaParse(GRAMMAR+FM+L, QIL, 'Input)),
         output : nil, default : MN, Atts >,
       QIL']
  if QIL =/= nil /\
     metaParse(GRAMMAR+FM+L, QIL, 'Input) : ResultPair .
```

Next the 'DATABASE-HANDLING' module must be extended with the *user-defined* functions that handle the $D_{\mathcal{L}}$ terms. As an example, consider the following rule in 'DATABASE-HANDLING'. It "detects" that Full Maude module was inserted by the user and correctly parsed by the 'GRAMMAR' signature into a term that begins—by matching against the pattern 'F[T, T']'—with either 'fmod_is_endfm' (a functional module), 'obj_is_endo' (another name for functional modules that comes from OBJ3 origins), 'obj_is_jbo' (yet another name for functional modules), 'mod_is_endm' (system modules), or omod_is_endom (object-oriented modules, not discussed here). It then proceeds to call the internal function 'procUnit' with the parsed term and the current database as arguments. Any user defined syntax must provide a rule similar to this one, that "detects" the successful parsing of terms of sort $D_{\mathcal{L}}$.

```
crl [module] :
   < O : X@Database | db     : DB,  input   : (F[T, T']),
                      output : nil, default : MN, Atts >
   => < O : X@Database |
          db : procUnit(F[T, T'], DB), input : nilTermList,
          output :
            ('Introduced 'module
              modNameToQid(parseModName(T)) '\n),
          default : parseModName(T), Atts >
   if  (F == 'fmod_is_endfm) or-else
      ((F == 'obj_is_endo)   or-else
      ((F == 'obj_is_jbo)    or-else
      ((F == 'mod_is_endm)   or-else
       (F == 'omod_is_endom)))) .
```

Chapter 5 describes how Full Maude was extended to support *Maude MSOS Tool*.

2.4 Modular Rewriting Semantics

In this Section we summarize the presentation of Modular Rewriting Semantics (MRS) as in [8, 44].

Modular rewriting semantics (MRS) [8, 44] is a technique for the modular specification of programming languages semantics in rewriting logic. An MRS specification is a rewrite theory developed according to some techniques that supports modular definitions. MRS specifications use a syntax-directed style of semantics, with program syntax being separated from semantic components, such as the environment, memory or synchronization signals.

Let us define more formally what does it mean for a specification to be modular [8]. When $\mathcal{L}_1$ is a language extension of $\mathcal{L}_0$, the first modularity requirement is *monotonicity*: there is a theory inclusion $\mathcal{R}_{\mathcal{L}_0} \subseteq \mathcal{R}_{\mathcal{L}_1}$. Monotonicity means that we do not need to retract earlier semantic definitions in a language extension.

A second modularity requirement is *ground conservativity*: for any ground Σ_0-terms $t, t' \in T_{\Sigma_i}$ (the set of K-kinded ground Σ_i terms) we have, (i) $E_0 \vdash t = t' \Leftrightarrow E_1 \vdash$

$t = t'$, (ii) $\mathcal{R}_{\mathcal{L}_0} \vdash t \to t' \Leftrightarrow \mathcal{R}_{\mathcal{L}_1} \vdash t \to t'$. Ground conservativity means that new semantic definitions do not alter the semantics of previous features on the previously defined language fragments. MRS defines then two techniques for the modular definition of programming language semantics, satisfying these two requirements.

The first modularity technique is *record inheritance*, which is accomplished through pattern matching modulo associativity, commutativity, and identity. Features added later to a language may necessitate adding new semantic components to the record; but the axioms of older features can be given once and for all: they will apply just the same with new components in the record (note that this technique is shared with MSOS: records in MRS are essentially labels in MSOS). The Maude specification of the equational theory of records is as follows.

```
fmod RECORD is
 sorts Index Component Field PreRecord Record .
 subsort Field < PreRecord .

 op null : -> PreRecord [ctor] .
 op _,_ : PreRecord PreRecord
          ->  PreRecord [ctor assoc comm id: null] .
 op _=_ : [Index] [Component] -> Field [ctor] .
 op {_} : [PreRecord] -> [Record] [ctor] .
 op duplicated : [PreRecord] -> [Bool] .

 var I : Index .
 vars C C' : Component .
 var PR : PreRecord .

 eq duplicated((I = C), (I = C'), PR) = true .

 cmb { PR } : Record if duplicated (PR) =/= true .
endfm
```

A ‘`Field`’ is defined as a pair of ‘`Index`’ and a ‘`Component`’; illegal pairs will be of kind ‘`[Field]`’. A ‘`PreRecord`’ is a possibly empty (‘`null`’) multiset of fields, formed with the union operator ‘`_,_`’ which is declared to be associative, commutative, and to have ‘`null`’ as its identity. Note the conditional membership defining a ‘`Record`’ as an “encapsulated” ‘`PreRecord`’ with no duplicated fields.

Record inheritance means that we can always consider a record with more fields as a special case of one with fewer fields. For example, a record with an environment component indexed by ‘`env`’ and a store component indexed by ‘`st`’ can be viewed as a special case of a record with just the environment component. Matching modulo associativity, commutativity, and identity supports record inheritance, because we can always use an extra variable ‘`PR`’ of sort ‘`PreRecord`’ to match any extra fields the record may have. For example, the function ‘`get-env`’ extracting the environment component can be defined by ‘`eq get-env(env = E:Env, PR:PreRecord) = E .`’ and will apply to a record with any extra fields that are matched by ‘`PR`’.

The second modularity technique is the systematic use of *abstract interfaces*. That is, the sorts specifying key syntactic and semantic entities are *abstract sorts* such that: (i) they only specify the *abstract functions* manipulating them, that is, a given *signature*, or *interface*, of abstract sorts and functions; *no axioms* are specified about such functions *at the level of abstract sorts*; (ii) in a language specification no *concrete* syntactic or semantic sorts are ever identified with abstract sorts: they are always either specified as *subsorts* of corresponding abstract sorts, or are mapped to abstract sorts by *coercions*; it is *only at the level of such concrete sorts* that *axioms* about abstract or auxiliary functions are specified.

Systematic use of the above two new techniques seems to ensure that the rewriting semantics of any language extension $\mathcal{L}_0 \subseteq \mathcal{L}_1$ is always modular provided that: (i) the only rewrite rules in the theories $\mathcal{R}_{\mathcal{L}_0}$ and $\mathcal{R}_{\mathcal{L}_1}$ are semantic rules

$$\langle f(t_1, \cdots, t_n), u \rangle \rightarrow \langle t', u' \rangle \Leftarrow C,$$

where C is the condition of the rule, f, is a language feature, e.g., ‘`if-then-else`’, u and u' are record expressions and u contains a variable ‘`PR`’ of sort ‘`PreRecord`’ standing for unspecified additional fields and allowing the rule to match by record inheritance; (ii) the following *information hiding* discipline should be followed in u, u', and any record expression appearing in C: besides any record syntax, only function symbols appearing in the *abstract interfaces* of some of the fields in the record can appear in record expressions; any auxiliary functions defined in concrete sorts of those field's components should never be mentioned; and (iii) the semantic rules of each programming language feature f should all be defined in the *same* theory, that is, either all are in $\mathcal{R}_{\mathcal{L}_0}$ or all in $\mathcal{R}_{\mathcal{L}_1}$.

MRS uses pairs, called *configurations*; the first component is the *program text*, and the second the *record* whose fields are the different *semantic entities* associated to a program's computation. We can specify configurations in Maude with the following membership equational theory:

```
fmod CONF
 is protecting RECORD .

 sorts Program Conf .

 op <_,_> : Program Record -> Conf [ctor] .
endfm
```

“Restricted configurations” are defined by the module ‘`RCONF`’, below. They are a means of controlling the rewrites at the conditions of MRS rules and are necessary to give the correct semantics for operational semantics transitions, as discussed on Section 2.4.1.

```
mod RCONF
 is extending CONF .

 op {_,_} : [Program] [Record] -> [Conf] [ctor] .
 op [_,_] : [Program] [Record] -> [Conf] [ctor] .
```

```
 vars P P' : Program .
 vars R R' : Record .

 crl [step] : < P , R > => < P' , R' >
 if { P , R } => [ P' , R' ] .
endm
```

As an example, let us specify the MRS of a very simple programming language that contains only an ML-like '`let-in-end`' construction and the '`sum`' function. The same language will be used on Section 4.5 to demonstrate the use of *Maude MSOS Tool*. We begin by defining its syntax using a functional module '`SIMPLE-LANGUAGE-SYNTAX`'. It imports modules '`EXP`' and '`ID`' that define respectively, sorts '`Exp`' (the *expressions* in the language) and '`Id`' (the identifiers). The two constructions of the language are defined as operators '`let_=_in_end`' and '`_sum_`'.

```
fmod EXP is
 sort Exp .
endfm

fmod ID is
 sort Id .
endfm

fmod SIMPLE-LANGUAGE-SYNTAX is
 protecting INT .
 protecting EXP .
 protecting ID .

 subsort Int < Exp .
 subsort Id < Exp .

 op let_=_in_end : Id Int Exp -> Exp .
 op _sum_ : Exp Exp -> Exp .
endfm
```

The specification of the '`let-in-end`' construction requires an environment for bindings. Following the modularity technique of using *abstract functions* to define components, we define the module '`BASIC-ENVIRONMENT`' with the abstract sort '`Env`', for environments. Furthermore, the following abstract functions are defined: '`find(ρ,i)`', which returns the value bound to identifier i in environment ρ, and '`override(ρ,i,n)`', which overrides environment ρ with the binding defined by the identifier i and the integer n. '`Env`' is declared as a component by being a subsort of '`Component`' and '`env`' is declared an index by being a constant of sort '`Index`'. The membership axiom at the end of the module defines the field formed by '`env`' and terms of sort '`Env`'.

```
fmod BASIC-ENVIRONMENT is
```

```
 extending RECORD .
 protecting INT .
 including ID .

 sorts Env .
 subsort Env < Component .

 op env : -> Index [ctor] .
 op find : Env Id -> [Int] .
 op override : Env Id Int -> Env .

 mb env = E:Env : Field .
endfm
```

The following module, 'SIMPLE-LANGUAGE-SEMANTICS', gives meaning to the 'let_=_in_end' and '_sum_' operators using a small-step operational semantics style similar to the rules specified in Section 4.5 and Section 2.1. In order to appear in configurations, we must subsort 'Exp' to 'Program'. Then, rules 'sum1', 'sum2', and 'sum3' work by evaluating first the left expression of the 'sum' operator, then the right, and, when both expressions were reduced to integers, evaluates to the "integer sum" of the values. Rule 'let1' evaluates the expression 'E' using an environment overridden, with the function 'override', with the binding defined by the 'X' and 'I'. Rule 'let2' tells that, when the expression reaches a final value, the entire expression is replaced by this value. Finally, rule 'find' specifies that the evaluation of an identifier 'X' is made first looking up its value bound to the environment 'Env'.

```
mod SIMPLE-LANGUAGE-SEMANTICS is
 protecting BASIC-ENVIRONMENT .
 protecting SIMPLE-LANGUAGE-SYNTAX .
 protecting RCONF .

 subsort Exp < Program .

 vars E1 E2 E'1 E'2 E E' : Exp .
 vars I1 I2 I I' : Int .
 vars Env Env' : Env .
 vars PR PR' : PreRecord .
 var  X : Id .
 vars R R' : Record .

 crl [sum1] : { E1 sum E2, R } => [ E'1 sum E2, R' ]
 if { E1, R } => [ E'1, R' ] .

 crl [sum2] : { I sum E2, R } => [ I sum E'2, R' ]
 if { E2, R } => [ E'2, R' ] .

 crl [sum3] : { I1 sum I2, R } => [ I, R ]
 if I := I1 + I2 .
```

```
 crl [let1] : { let X = I in E end, { (env = Env), PR } }
     => [ let X = I in E' end, { (env = Env), PR' } ]
 if Env' := override (Env, X, I) /\
    { E, { (env = Env'), PR } }
    => [ E', { (env = Env'), PR' } ] .

 rl [let2] : { let X = I in I' end, R }
    => [ I', R ] .

 crl [find] : { X, { (env = Env), PR } }
     => [ I, { (env = Env), PR } ]
 if I := find (Env, X) .
endm
```

At this point the specification is not executable since we must provide a concrete implementation for the abstract functions defined in the module 'BASIC-ENVIRONMENT'. Let us show a possible implementation of the module 'CONCRETE-ENVIRONMENT' in which the actual bindings are constructed with the operator '_|->_' and the environment is an associative-commutative juxtaposition operator of sort 'CEnv'. The coercion function '<_>' moves the terms of sort 'CEnv' to 'Env', provided that there are no duplicated bindings in its argument.

```
fmod CONCRETE-ENVIRONMENT is
 including BASIC-ENVIRONMENT .
 protecting INT .

 sort Bind CEnv .
 subsort Bind < CEnv .

 op void : -> CEnv [ctor] .
 op _|->_ : Id Int -> Bind [ctor] .
 op __ : CEnv CEnv -> CEnv [ctor assoc comm id: void] .
 op <_> : [CEnv] -> [Env] .
 op dupl : [CEnv] -> [Bool] .

 var  X : Id .
 vars I I' I1 : Int .
 var  CE : CEnv .

 eq dupl((X |-> I) (X |-> I1) CE) = true .
 cmb < CE > : Env if dupl(CE) =/= true .

 eq find(< (X |-> I) CE >, X) = I .

 eq override(< (X |-> I) CE >,X,I') =
    < (X |-> I') CE > .
 eq override(< CE >, X, I') =
```

```
    < (X |-> I') CE > [owise] .
endfm
```

The module 'SIMPLE-LANGUAGE' gathers all modules into a single specification.

```
mod SIMPLE-LANGUAGE is
 including SIMPLE-LANGUAGE-SYNTAX .
 including SIMPLE-LANGUAGE-SEMANTICS .
 including CONCRETE-ENVIRONMENT .
endm
```

To execute a program in our language, we create constants 'x' and 'y', of sort 'Id', and execute the configuration with the program text and the record, containing an empty environment, using Maude's 'rewrite' command.

```
rewrite in SIMPLE-LANGUAGE :
  < let x = 10 in
     let y = 10 in x sum y end
    end,{env = < void >} > .
rewrites: 108 in 1ms cpu (10ms real) (108000 rewrites/second)
result Conf: < 20,{env = < void >} >

Bye.
```

2.4.1 Modular Rewriting Semantics and MSOS

Modular Rewriting Semantics has a close relation with Mosses's Modular Structural Operational Semantics (MSOS) [53]. This is due to the fact that structural operational semantics has a direct representation in rewriting logic [38, 6, 68, 44, 8] and that Modular Rewriting Semantics and MSOS share a technique to achieve modularity based on the encapsulation of the semantic information, called record inheritance in MRS. In [43, 44] Braga and Meseguer propose a semantics-preserving transformation from MSOS to Modular Rewriting Semantics with a formal proof of the bisimulation between the models of MSOS and Modular Rewriting Semantics, which we discuss in this section.

The use of label expressions in MSOS such as $\{\rho = \rho_0, \tau' = \tau_0, \ldots\}$, as we mentioned on Section 2.1 is similar to Standard ML's pattern matching for records. This has a close relationship to the *record inheritance* technique of Modular Rewriting Semantics where the notation "..." is equivalent to the use of a metavariable 'PR' of sort 'PreRecord' that matches against any unspecified components.

Recalling the general form of MSOS rules from Section 2.1:

$$\frac{c_1, \ldots, c_n}{c}$$

where each condition c_i is either a conditional transition $v_i -\alpha_i\rightarrow v_i'$ or a predicate p_i, and α_i is label. The conclusion is a transition $t -\alpha\rightarrow t'$, where t, t', v_i, v_i' are value-added syntax trees that belong to a sorts that are subsorts of 'Program'. Labels in MSOS are easily represented by records of the sort 'Record' in Modular Rewriting Semantics using a metavariable of the sort 'PreRecord' to represent unspecified components on the label. We also need a sort 'IRecord' to represent *unobservable labels* and 'IPreRecord' to represent unspecified components of unobservable labels, where 'IRecord' $<$ 'Record' and 'IPreRecord' $<$ 'PreRecord' and a partial composition operator '_;_' over 'Record'. In the remainder of this Section let us use metavariables X, X' of sort 'Record', variables U, U' of sort 'IRecord', variables PR, PR' of sort 'PreRecord', and variables UPR, UPR' of sort 'IPreRecord'.

Furthermore, it is necessary to understand the semantics of transitions at the conditions of MSOS rules. Unlike rewriting logic, MSOS transitions at the conditions are *one step conditions*, whereas, rewrites at the conditions in rewriting logic are *zero or more steps conditions* due to the deduction rules shown in Section 2.2. In order to correctly simulate transitions at the MSOS conditions we need a way of controlling rewrites at the conditions. One way of doing this is using a "restricted configuration," as defined by the module 'RCONF' in Section 2.4.

An MSOS specification $\mathcal{M}$ also defines the abstract syntax of a programming language $\mathcal{L}$. We shall assume, in this Section, that this syntax, and any semantic components needed, are specified as described in Section 2.4, that is, using a membership equational theory $(\Omega_{\mathcal{L}}, E_{\mathcal{L}})$. We further assume that MSOS transitions have a representable notation (by defining a *normal form*, see below) defined by a signature $(\Omega_{\mathcal{M}}, E_{\mathcal{M}})$. Let (Ω, E) be a membership equational theory that includes both $(\Omega_{\mathcal{L}}, E_{\mathcal{L}})$ and $(\Omega_{\mathcal{M}}, E_{\mathcal{M}})$. An MSOS specification $\mathcal{M}$ is, then, a rewriting logic theory $\mathcal{R} = (\Omega, E, R)$ where each transition rule is represented by a rewrite rule in R. The transition from MSOS to Modular Rewriting Semantics is then a mapping between rewrite theories:

$$\tau : (\Omega, E, R) \rightarrow (\Omega', E', R')$$

where (Ω', E', R') includes the 'RECORD' (with the extra sorts 'IRecord' and 'IPreRecord') and 'RCONF' and the transition rules R' are obtained from R as follows. First we assume that the rules in R are in a *normal form*, that is:

- predicates in the conditions do not involve record, field, or index expressions in their arguments;
- a record expression appearing in the conditions or in the conclusion is either: variables X or U; a constructor term of the general form $\{i_1 = w_1, \ldots, i_n = w_n, PR\}$ or $\{i_1 = w_1, \ldots, i_n = w_n, UPR\}$, with $n \geq 0$, where indices i_i are constants of the sort 'Index' and may be primed and w_i are components whose sort is a subsort of 'Component' and there is a membership assertion $i_i = w_i : \texttt{Field}$.

As an example, consider rule 2.14, on Section 2.1. The rule could be rewritten as:

$$\frac{\alpha = \{env = \rho[m/x], \ldots\} \quad e_1 -\alpha\rightarrow e_1'}{\texttt{let } x\texttt{=}m \texttt{ in } e_1 \texttt{ end } -\{env = \rho, \ldots\}\rightarrow \texttt{ let } x\texttt{=}m \texttt{ in } e_1' \texttt{ end}} \quad (2.17)$$

Rule 2.17 is not in the normal form since it has a condition involving a record. Rule 2.14 is in the desired normal form. Even though there is no formal proof that the normal form requirement is not too restrictive, our experience shows that indeed it is not.

The mapping τ is defined as follows (taken from [44]):

- (Ω', E') is obtained from (Ω, E) by:
  - omitting all the primed indices and their related equation and membership axioms from the record subspecification, and adding the unprimed version of each write-only index;
  - defining sorts '`ROPreRecord`', '`RWPreRecord`', '`WOPreRecord`' (all containing the constant '`null`') of the sort '`PreRecord`', corresponding to those parts of the record involving read-only, read-write, and write-only fields. Let us use metavariables A, B, and C to range over those respective sorts, with their primed variants A', B', C';
  - in '`WOPreRecord`' we also axiomatize a prefix predicate $\sqsubseteq$, where $C \sqsubseteq C'$ means that for each write-only field k the string $C.k$ is a prefix of the string $C'.k$;
  - adding the signature of the module '`RCONF`'
- R' contains the '`step`' rule in '`RCONF`', and for each MSOS rule in R (in the normal form), the corresponding rewrite rule of the form:

$$\{t, u^{pre}\} \to [t', u^{post}] \Leftarrow \{v_1, u_1^{pre}\} \to [v_1', u_1^{post}] \wedge \cdots \\ \cdots \wedge \{v_n, u_n^{pre}\} \to [v_n', u_n^{post}]$$

  where for each record expression u in the MSOS rule, u^{pre} and u^{post} are defined as follows:

  - for u a record expression of the general form X or $\{PR\}$, u^{pre} is a record expression of the form $\{A, B, C\}$, and u^{post} has the form $\{A, B', C'\}$;
  - for u a record expression of the general form U or $\{UPR\}$, u^{pre} is a record expression of the general form R or $\{PR\}$ and $u^{post} = u^{pre}$;
  - for u a record expression of the general form $\{i_1 = w_1, \ldots, i_n = w_n, PR\}$, with $n \geq 1$, u^{pre} and u^{post} are record expressions similar to u where: (i) if a read-only field expression $i = w$ appears in u, then it appears in u^{pre} and u^{post}; (ii) if a write-only field expression $i' = w$ appears in u, then u^{pre} contains a field expression of the form $i = l$, with l a new list variable in the corresponding data type, and u^{post} contains a field expression of the form $i = l \cdot w$ (if u labels a condition, u^{pre} contains $i = \varepsilon$, where ε is the identity of the list data type, and u^{post} contains $i = w0$; (iii) if a read-write pair of field expressions $i = w$, $i' = w'$ appear in u then u^{pre} contains $i = w$ and u^{post} contains $i = w'$; (iv) PR is translated in u^{pre} as A, B, C, and in u^{post} as A, B', C';
  - for u a record expression of the general form $\{i_1 = w_1, \ldots, i_n = w_n, UPR\}$, with $n \geq 1$, then cases (i)–(iii) also apply here and (iv) UPR is translated in both u^{pre} and u^{post} as PR;

- any conditions that are not transitions are moved unaltered to the rule
- the condition is augmented with $C \sqsubseteq C'$ if expressions of the form A, B, C and A, B', C' were introduced in the terms u^{pre} and u^{post} in the conclusion.

As an example rule 2.14 is translated into the following:

$$
\begin{aligned}
&\{\ \texttt{let}\ x = m\ \texttt{in}\ e_1\ \texttt{end}, \{env = \rho, PR\}\} \\
&\rightarrow [\ \texttt{let}\ x = m\ \texttt{in}\ e_1'\ \texttt{end}, \{env = \rho, PR\}] \\
&\Leftarrow \{e_1, \{env = \rho[m/x], PR\}\} \rightarrow [e_1', \{env = \rho[m/x], PR\}]
\end{aligned}
$$

Chapter 3

Related work

Although operational-style specifications of programming languages may be defined using conventional programming languages and other formal tools, we opted to describe here tools that have the specific purpose of supporting operational semantics specifications. In light of this, we opted to analyze in this chapter Hartel's LETOS [27], Pettersson's RML [56] and Mosses's MSOS Tool [52], as three significant examples of operational semantics tools.

We begin by describing LETOS, "A Lightweight Execution Tool for Operational Semantics." The tool is written in C with a lex and yacc parser and uses a superset of Miranda [66] to specify operational and denotational semantics specifications, that are converted into Miranda scripts (although the author mentions that, with minor changes, the output could be changed instead to Haskell [32]). An additional feature of LETOS is its support for pretty-printing specifications in LaTeX and to provide execution tracing using HTML pages. LETOS has partial support for non-deterministic specifications, simulated by functions returning lists, and the final result of a non-deterministic specification will be always only one of the several possible final values. Abstract syntax is specified using Miranda's user-defined data type syntax. As is usual in operational semantics specifications [54], LETOS allows the definition of several different relations between configurations.

Let us exemplify all these characteristics using a simple example, based on the semantics of the Mini-Freja [56] language. A LETOS specification is actually a LaTeX document in which the operational semantics is separated from normal text by markers '`.MS`' and '`.ME`'. For example, the following fragment specifies the abstract syntax of expressions in the Mini-Freja language. Even though the abstract syntax is expressed in prefix format, infix functions may also be specified.

```
\[
\begin{array}{@{}l@{}ll}
.MS
macro_Exp ::=
    CONSTexp macro_Const |
    VARexp macro_Var |
    CONSexp (macro_Exp, macro_Exp) |
    LAMexp (macro_Var, macro_Exp) |
    PRIMONEexp (macro_Primone, macro_Exp) |
```

```
    PRIMTWOexp (macro_Primtwo, macro_Exp, macro_Exp) |
    IFexp (macro_Exp, macro_Exp, macro_Exp) |
    APPexp (macro_Exp, macro_Exp) |
    CASEexp (macro_Exp, [macro_Rule]) |
    RECexp ([(macro_Var, macro_Exp)], macro_Exp) ;
.ME
\end{array}
\]
```

which is typeset into the following:

$$\begin{array}{ll} \text{Exp}\equiv & \text{CONSTexp Const} \mid \\ & \text{VARexp Var} \mid \\ & \text{CONSexp(Exp, Exp)} \mid \\ & \text{LAMexp(Var, Exp)} \mid \\ & \text{PRIM1exp(Prim1, Exp)} \mid \\ & \text{PRIM2exp(Prim2, Exp, Exp)} \mid \\ & \text{IFexp(Exp, Exp, Exp)} \mid \\ & \text{APPexp(Exp, Exp)} \mid \\ & \text{CASEexp(Exp, [Rule])} \mid \\ & \text{RECexp([(Var, Exp)], Exp);} \end{array}$$

The following fragment (already typeset) specifies a particular type of relation, '`ifchoose`' that selects either expression, based on the boolean value of its first argument. It is used on the semantics of the "if" construction of Mini-Freja that follows.

$$\begin{array}{lll} \overset{\text{ifchoose}}{\Rightarrow} & :: & ((\mathsf{bool},\ \text{Exp},\ \text{Exp})\leftrightarrow\text{Exp}); \\ [\mathsf{ifchoose}^{\mathsf{true}}] & & (\mathsf{True},\ \mathsf{e2},\ \mathsf{e3}) \overset{\text{ifchoose}}{\Rightarrow} \mathsf{e2}; \\ [\mathsf{ifchoose}^{\mathsf{false}}] & & (\mathsf{False},\ \mathsf{e2},\ \mathsf{e3}) \overset{\text{ifchoose}}{\Rightarrow} \mathsf{e3}; \end{array}$$

As an example of using a different relation in a transition, consider the semantics of the "if" statement of the Mini-Freja language, given using an *eval* relation that uses the *ifchoose* relation.

$$\begin{array}{lll} \overset{\text{eval}}{\Rightarrow} & :: & ((\text{Env},\ \text{Exp})\leftrightarrow\text{Val}); \\ & & \dfrac{(\mathsf{rho},\ \mathsf{e1}) \overset{\text{eval}}{\Rightarrow} \text{CONSTval(BOOLcnst } \mathsf{flag}), \quad (\mathsf{flag},\ \mathsf{e2},\ \mathsf{e3}) \overset{\text{ifchoose}}{\Rightarrow} \mathsf{e}, \quad (\mathsf{rho},\ \mathsf{e}) \overset{\text{eval}}{\Rightarrow} \mathsf{v}}{(\mathsf{rho},\ \text{IFexp}(\mathsf{e1},\ \mathsf{e2},\ \mathsf{e3})) \overset{\text{eval}}{\Rightarrow} \mathsf{v};} \\ [\mathsf{eval}^{\mathsf{if}}] & & \end{array}$$

The next system analyzed is Pettersson's Relational Meta-Language (RML), which provides support for natural semantics [33] specifications. The RML system is a compiler written in Standard ML that compiles RML into efficient low-level C code. Like LETOS,

RML supports creating different relations to be used on transitions. Unlike LETOS, RML does not have support for pretty-printing of specifications or tracing executions. As an example of RML specifications, let us see the equivalent fragment of Mini-Freja that was shown above. We begin with the abstract syntax of expressions. RML does not allow the definition of infix functions, so the syntax is given only in a prefix-style notation.

```
datatype exp = CONSTexp of const
             | VARexp of var
             | CONSexp of exp * exp
             | LAMexp of var * exp
             | PRIM1exp of prim1 * exp
             | PRIM2exp of prim2 * exp * exp
             | IFexp of exp * exp * exp
             | APPexp of exp * exp
             | CASEexp of exp * crule list
             | RECexp of (var * exp) list * exp
```

The relation '`if_choose`' below has the same meaning as ifchoose on the LETOS specification:

```
relation if_choose: (bool, Absyn.exp, Absyn.exp) => Absyn.exp =
 axiom if_choose(true, e2, _) => e2
 axiom if_choose(false, _, e3) => e3
end
```

Finally, we show the transition rule for the "if" construction.

```
rule eval(rho, e1) => CONSTval(Absyn.BOOLcnst flag) &
     if_choose(flag, e2, e3) => e &
     eval(rho, e) => v
     ----------------
     eval(rho, Absyn.IFexp(e1,e2,e3)) => v
```

The final tool analyzed is Mosses's MSOS Tool, written in Prolog, which converts MSOS specifications to Prolog. The MSOS Tool supports both small-step and big-step styles of operational specification and its input language is MSDF, the *Modular SOS Definition Formalism*. The language described in Chapter 4 is actually based on MSOS Tool's MSDF specification language. The version of the MSOS Tool described here is based on [52].

Unlike LETOS and RML, MSDF does not allow the definition of different relations. Only two predefined relations are available: '`--->`', used in small-step dynamic semantics, and '`===>`', used in static semantics and big-step dynamic semantics. Abstract syntax is given using a BNF-like notation similar to LETOS, but only prefixed functions may be defined. The MSOS Tool is tightly bound to the Constructive MSOS (see Section 6.1) style of specifying programming languages semantics and the tool comes with a vast library of reusable modules providing the semantics of several different basic constructions. Mosses's

notes ([52]) defines the mapping from the concrete syntax of languages such as bc and ML using Prolog's *Definite Clause Grammars* (DCG).

Let us demonstrate the MSOS Tool by showing as example one of the modules present in the tool library. Each module is specified using three different files, organized on a directory structure. The directory 'Cons' contains the abstract syntax and semantics of several basic programming languages constructs. Inside this directory, 'Cmd' contains the constructions related the imperative facet of programming languages. We selected the directory 'cond-nz' which contains a conditional command. We begin by presenting the abstract syntax, given on a file named 'ABS.msdf'. Module dependencies are automatically provided by the tool: in this example, the user does not have to worry about having to include the modules that define the sets 'Cmd', and 'Exp'.

```
Cmd ::= cond-nz(Exp,Cmd)
```

Static semantics for the function is specified in a module named 'CHK.msdf':

```
ValueType ::= int

  E ===> int, C ===> void
--------------------------
cond-nz(E,C):Cmd ===> void
```

Dynamic semantics is specified in a module named 'RUN.msdf':

```
Value ::= Integer

               E --{...}-> E'
----------------------------------------
cond-nz(E,C):Cmd --{...}-> cond-nz(E',C)

cond-nz(0,C):Cmd ---> skip

        N \= 0
-----------------------
cond-nz(N,C):Cmd ---> C
```

The semantics of several programming languages are provided as examples in the directory 'Lang'. For example, the following Prolog code demonstrate the use of DCG to define the translation from bc to 'cond-nz':

```
stmt(cond_nz(E,C)) -->
        "if", i, "(", i, expr(E), i, ")", i, stmt(C).
```

Unlike LETOS, and like RML, MSOS Tool does not have support for the pretty-printing of specifications, but does have trace capabilities.

Chapter 4

Maude MSOS Tool

This Chapter describes the *Maude MSOS Tool* (MMT), a programming environment for Modular SOS specifications. MMT is a formal tool in the sense of [15], implemented as a conservative extension of *Full Maude*, that compiles MSOS specifications into rewriting logic. The implementation is based on a mapping that was first described in [6] and later developed in [8]. The compilation into rewriting logic modules enables the use of Maude's execution and formal verification tools in MSOS specifications.

The syntax of modules accepted by MMT is based on the *Modular SOS Definition Formalism* (MSDF) described by Mosses in [52]. Both languages are similar enough to warrant the designation of the specification language of the MMT also as "MSDF." The minor differences, described in Chapter 7, that exist between the two are restricted to idiosyncrasies of the Maude parser. The "MSDF" designation henceforth refers to the version currently accepted by the MMT, except otherwise noted.

While MMT was designed with the primary objective of being a formal environment for the specification of programming languages, operational semantics, including MSOS, is an environment applicable on a wide variety of topics. For example, operational semantics was used in the development of several concurrency calculi, such as CCS [50] and the π-calculus [51]. However, given its main purpose, we opted for the exposition of the different MSDF constructions by motivating their use on the formal definition of some programming language $\mathtt{L}$.

To give the definition of $\mathtt{L}$ is, in fact, to give the definition for each one of its $\mathtt{L}_\mathtt{c}$ constructions. It consists of the formal description of the *syntax* and *semantics* of $\mathtt{L}_\mathtt{c}$. For the syntax we follow the traditional approach and avoid the complications of concrete syntax by adopting an abstract version of $\mathtt{L}_\mathtt{c}$ (see, for example, [58, Section 1.4.1] and [52, Section 1.2.3] for a discussion on this). Let us call this abstract construction $\mathcal{L}_\mathtt{c}$. Also following usual practices, we describe the context-free grammar of $\mathcal{L}_\mathtt{c}$ using Backus Naur Form (BNF) notation.[1] The specification of a programming language abstract syntax is the subject of Section 4.2.2.

The semantics of the $\mathcal{L}_\mathtt{c}$ construction is defined by *transition rules*, with *labels* containing the necessary semantic components, as discussed on Section 2.1. We discuss how labels and transitions are specified in MSDF in Sections 4.2.3 and 4.2.4, respectively.

[1] An interesting historical perspective on the name Backus Naur Form versus Backus Normal Form appears on Donald Knuth's collection of programming language papers [35].

Specifications may be organized into modules to allow the modular construction of a specification, something desirable not only for didactic purposes, but also as a sound engineering practice to cope with the complexity inherent on any large scale specification project. The use of modules is described on Section 4.2.1.

4.1 Notational conventions

On the following sections the grammar of MSDF constructions is specified using an extended BNF grammar, with terminals '`in teletype font and between single quotes`' and non-terminals specified in ⟨ sans-serif font and between angle brackets ⟩. Extended BNF notation is always used on an expression within meta-parentheses: '$(E)^*$' and '$(E)^+$' denote, respectively, zero or more, and one or more, repetitions of E; '$(E)^?$' denotes an optional expression.

Before we explain the grammar of MSDF's constructions we must discuss some lexical aspects. As an extension of *Full Maude*, we also make use of Maude's `LOOP-MODE` (see Section 2.3), and are restricted to its lexical analysis capabilities. In order to insulate this description from the technical details of Maude's own current lexical analysis, we use some predefined non-terminals in an abstract way: the first is ⟨ id ⟩, for *identifiers*, that follows Maude's definition of identifiers. As of Maude version 2.1.1 the lexical rules for identifiers are as follows (taken from [14]):

> Any finite string of ASCII characters such that:
>
> - it does not contain any white space;
> - the characters '`{`', '`}`', '`(`', '`)`', '`[`', '`]`', and '`,`' break a sequence of characters into several identifiers;
> - the back-quote character '`` ` ``' is used as an escape character to indicate that a blank space or the special characters following it do not break the sequence.

Identifiers with the initial letter in uppercase are represented by ⟨ upper-id ⟩, while those with the opposite constraint—the initial letter in lowercase—are represented by ⟨ lower-id ⟩. With the current version of Maude it is not possible to specify this and other (e.g., that an identifier may consists only of letters) restrictions, usually stated as regular expressions, so they must be checked after the parsing is done.

Another predefined non-terminal is ⟨ term ⟩ which represents *value added syntax trees* that appear in transition rules.

In Full Maude, all user input must be made between parentheses.

4.2 MSDF syntax

These Sections describe the elements of the MSDF specification language. Each Section begins with a formal exposition of the constructions, followed by an illustrative example, and ends with the details on usage.

4.2.1 Modules

⟨ module ⟩ → ‘msos’ ⟨ id ⟩ ‘is’ (⟨ see ⟩)* (⟨ declaration ⟩)* ‘sosm’
⟨ see ⟩ → ‘see’ ⟨ name ⟩ (‘,’ ⟨ name ⟩)* ‘.’

All modules in MSDF begin with the string ‘msos’ and end with ‘sosm’. An MSDF module name is any identifier (⟨ id ⟩) accepted by Maude (Section 4.1). To include other modules, one uses the ⟨ see ⟩ constructions, which is the keyword ‘see’, followed by a list of modules identifiers, separated by commas; multiple ⟨ see ⟩ lines are permitted for flexibility of the specification text, but they are only sugar for a single line of module importations.

The remainder of the module consists of a sequence of MSDF declarations, represented by the non-terminal ⟨ declaration ⟩. In order to allow a flexible specification, the order does not matter, since all declarations are first collected before the compilation begins.

As an example of module inclusion, consider the following MSDF module, that defines a module ‘PL-SYNTAX’ that, in turn, includes the modules that define the several different components of some fictitious language PL.

```
msos PL-SYNTAX is
 see PL-EXPRESSIONS, PL-DECLARATIONS .
 see PL-IMPERATIVES, PL-ABSTRACTIONS .
 see PL-CONCURRENCY .
sosm
```

As a general rule, in order to be used in a module an MSDF declaration must be previously defined in some other module (or in the module itself) and its declaring module must be explicitly included. However, it so happens that some inclusions may be better omitted not only for brevity, but also for clarity of MSDF specifications. It is not uncommon in programming language definitions for a basic set of modules to be needed in *most* of the other modules, since primitive constructions, such as commands and expressions are probably used by even the most advanced features; it would be tedious and error prone to force the user to explicitly declare these inclusions on each module that they are needed, specially when they are obviously needed.

The concept of “obvious need” is subjective and, to avoid confusion, the MMT has a simple rule for the omission of a module inclusion: all defining modules of sets that are used in a module are included by default. This includes not only sets used on the datatype definition part, but also the label declaration part.

For example, consider the following MSDF module that defines an abstract syntax of conditional expressions:

```
msos COND is
 Exp ::= if Exp then Exp else Exp .
sosm
```

The system automatically includes the module that contains the declaration of the set ‘`Exp`’. If this module is, say, ‘`EXP`’, then the expanded module, without any implicit inclusions, would be:

```
msos COND is
 see EXP .

 Exp ::= if Exp then Exp else Exp .
sosm
```

It is important to emphasize that this rule is restricted to *sets* only; if a module defines a construction that uses other constructions, the defining modules must be explicitly included. Consider the following example, where a ‘`while`’ is defined in terms of conditionals (‘`if then else`’), and command sequencing (‘`;`’). The modules that define these constructions (‘`COND`’, and ‘`SEQ`’, respectively) had to be explicitly included, while the modules that define the sets ‘`Exp`’ and ‘`Cmd`’ were omitted.

```
msos WHILE is
  see COND, SEQ .

  Cmd ::= while Exp do Cmd .

  (while Exp do Cmd) : Cmd -->
   if Exp then (Cmd ; while Exp do Cmd) else skip .
sosm
```

4.2.2 Datatype definitions

⟨ setid ⟩ → ⟨ upper-id ⟩
⟨ opid ⟩ → ⟨ lower-id ⟩
⟨ dsetid ⟩ → ⟨ setid ⟩ | ⟨ setid ⟩‘`*`’ | ⟨ setid ⟩‘`+`’
⟨ mixfix-fun ⟩ → ⟨ opid ⟩ | ⟨ dsetid ⟩
⟨ dec ⟩ → ⟨ setid ⟩ ‘`.`’ | ⟨ setid ⟩ ‘`::=`’ ⟨ dec-rhs ⟩ (‘`|`’ ⟨ dec-rhs ⟩)* ‘`.`’
⟨ dec-rhs ⟩ → ⟨ dsetid ⟩ | ⟨ mixfix-fun ⟩$^+$ (‘`[`’⟨ attr ⟩‘`]`’)$^?$
⟨ param ⟩ → ⟨ dsetid ⟩ | ‘`(`’⟨ dsetid ⟩‘`)List`’ | ‘`(`’⟨ dsetid ⟩‘`)Set`’
    | ‘`(`’⟨ dsetid ⟩‘`,`’⟨ dsetid ⟩‘`)Map`’
⟨ equiv ⟩ → ⟨ setid ⟩ ‘`=`’ ⟨ param ⟩ ‘`.`’
⟨ declaration ⟩ → ⟨ dec ⟩ | ⟨ equiv ⟩

Although specified using BNF notation, the abstract syntax definition of a programming language in MSDF is in fact an algebraic datatype definition. This is also true in other operational semantics tools, including Mosses's own MSOS Tool, in Prolog and Hartel's LETOS, in Miranda. In this sense, non-terminals are sets, and sequences of non-terminals and terminals (⟨ mixfix-fun ⟩$^+$) are n-ary functions specified in the so-called

mixfix form with the non-terminals representing the arguments and the terminals representing the function name in mixfix form. A function that contains only a single terminal is called a *constant*.

In MSDF, a ⟨ setid ⟩ is a *primitive set* name, an uppercase identifier (⟨ upper-id ⟩), such as '`Integer`'. This identifier must contain only letters, due to restrictions related to the formation of metavariables (see Section 4.2.4); a ⟨ dsetid ⟩ is a *derived set* name, since it represents a set automatically derived from the primitive sets. There are two possible derived sets: the set of possible-empty sequence of elements and the set of non-empty sequence of elements. For some set `s`, the former is written as `s*` while the later as `s+`.

The non-terminal ⟨ dec ⟩ declares either a new set, a subset inclusion declaration, or a function declaration, depending on the right-hand side (⟨ dec-rhs ⟩). If the right-hand side is a single ⟨ dsetid ⟩, then it is a subset inclusion (e.g., '`Exp ::= Value .`'). On the other hand, if the right-hand side is a sequence of lowercase and uppercase identifiers, it is interpreted as a mixfix function, as described above, e.g., '`Exp ::= if Exp then Exp else Exp .`'. A prefix form is, of course, a particular case of a mixfix function, e.g., '`Sys ::= parallel (Cmd, Cmd)`'. The optional attributes that may be specified to a mixfix function (the non-terminal ⟨ attr ⟩ enclosed between bracket parentheses) are discussed at the end of this Section.

Currently, MMT predefines two built-in sets: integers ('`Int`'), and booleans ('`Boolean`'), and their respective sequences '`Int*`', '`Int+`', '`Boolean*`', and '`Boolean+`'. The set '`Boolean`' contains two constants '`tt`' and '`ff`' that represents the values true and false, respectively.

Parameterized sets, defined by the non-terminal '`param`', are obtained using a modifier over one or two sets: given a set `s`, '`(s)List`' is the set of lists of elements of `s`; '`(s)Set`' is the set of finite sets of elements of `s`; given a second set `k`, '`(s,k)Map`' is the set of finite mappings from elements of set `s` to elements of `k`. Parameterized sets are not to be used directly, but by defining *equivalent sets* using the syntax of the non-terminal '`equiv`'.

The following fragment exemplifies the use of datatype definitions. Four sets are declared: '`Exp`', '`Value`', '`Cmd`', and '`Id`'. These sets represent, respectively, the expressions, values, commands, and identifiers of some programming language. Next, a subset inclusion is defined between '`Value`', and the built-in sets '`Int`' and '`Boolean`', that is, the set of values is augmented with elements from the sets of integers and booleans. Following, the abstract syntax of two constructions is defined in mixfix form: a conditional expression, and a looping command, which expects to receive a non-empty sequence of commands as its body. Finally, an equivalence between the set '`Env`' and the set '`(Id, Value)Map`'—a mapping between identifiers and values, traditionally used as an environment for bindings—is defined.

```
Exp .  Value .
Cmd .  Id .
Value ::= Int | Boolean .
Exp ::= if Exp then Exp else Exp .
Cmd ::= while Exp do Cmd+ .
Env = (Id, Value)Map .
```

Finally, mixfix functions in MSDF may have *attributes* to simplify notation. They are currently the attributes supported by the Maude system: '`associative`' (abbreviated '`assoc`'), used to define an associative binary function; '`commutative`' (abbreviated '`comm`'), used to define a commutative binary function; '`precedence` n' (abbreviated '`prec` n') that gives the precedence value of that function, where n is an integer between zero and $2^{31} - 1$, where a lower value indicates a tighter binding; and '`identity:t`' (abbreviated '`id:t`') that establishes the term `t` as the identity to the specified function. As in Maude, these attributes may be combined.

Attributes in MSDF datatype definitions permit the creation of a more flexible abstract syntax notation, while keeping its simplicity. The use of precedence values, for example, permits the specification of arithmetic functions with the right grouping on the abstract syntax, something that is usually done on the concrete syntax and then moved on to the abstract syntax explicitly. The Java Language Specification [26], for example, defines non-terminals '`UnaryExpression`', '`MultiplicativeExpression`', '`AdditiveExpression`', and '`RelationalExpression`' to cope with the several levels of precedence of these different expressions. Using MSDF, one may specify the same requirements using precedence and associative attributes as follows:

```
Exp .
Exp ::= Exp + Exp [assoc prec 10] .
Exp ::= Exp - Exp [assoc prec 30] .
Exp ::= Exp * Exp [assoc prec 20] .
Exp ::= Exp / Exp [assoc prec 20] .
Exp ::= Exp < Exp [assoc prec 10] .
Exp ::= Exp > Exp [assoc prec 10] .
Exp ::= Exp == Exp [assoc prec 10] .
```

By asking Maude to parse the expression '`e1 + e2 * e3 * e4 * e5 - e6`' using the declarations above, we obtain the following term, shown with parentheses to explicit the grouping order: '`(((e1 + e2) * (e3 * (e4 * e5))) - e6)`'.

4.2.3 Labels

⟨ label ⟩ → '`Label={`' ⟨ field ⟩ ('`,`' ⟨ field ⟩)* '`, ...} .`'
⟨ field ⟩ → ⟨ index ⟩ '`:`' ⟨ derived ⟩
⟨ index ⟩ → ⟨ lower-id ⟩
⟨ declaration ⟩ → ⟨ label ⟩

Each MSDF module may contain at most one label declaration using the syntax of the non-terminal ⟨ label ⟩. A label consists of a sequence of type declarations of ⟨ field ⟩. Each field consists of an ⟨ index ⟩ (which is a lowercase identifier, ⟨ lower-id ⟩) and the type of its component (⟨ derived ⟩).

The indices of the components defines a field to be read-only, read-write, or write-only: if there is a single, unprimed index, then the field defines a read-only component, as in:

```
Label = { env : Env, ... }
```

An index that appears both unprimed and primed defines a read-write component. Both components must be of the same type.

```
Label = { st : Store, st' : Store, ... }
```

Finally, a single, primed, index defines a write-only component. The only admissible type of write-only components are *sequences* of primitive sets, as in:

```
Label = { output' : Value*, ... }
```

If there are multiple label declarations in a single module, the last declaration is taken into account, while the others are ignored. Of course, a single label declaration may define several fields:

```
Label = { env : Env, st : Store, st' : Store,
          output' : Value*, ... }
```

4.2.4 Semantic transitions

⟨ transition ⟩ → ⟨ cond-transition ⟩ | ⟨ uncond-transition ⟩
⟨ cond-transition ⟩ → ⟨ cond ⟩ (‘,’ ⟨ cond ⟩)* (‘[’ ⟨ label ⟩ ‘]’)$^?$

quad ‘--’ ⟨ step ⟩ ‘.’
⟨ uncond-transition ⟩ → (‘[’ ⟨ label ⟩ ‘]’)$^?$ ⟨ step ⟩ ‘.’
⟨ label ⟩ → ⟨ id ⟩
⟨ typed-term ⟩ → ⟨ term ⟩ ‘:’ ⟨ dsetid ⟩
⟨ step ⟩ → ⟨ typed-term ⟩ ⟨ relation ⟩ ⟨ term ⟩
⟨ cond-step ⟩ → ⟨ term ⟩ ⟨ relation ⟩ ⟨ term ⟩
⟨ relation ⟩ → ‘-->’ | ‘==>’ | ‘-’ ⟨ label-exp ⟩ ‘->’ | ‘=’ ⟨ label-exp ⟩ ‘=>’
⟨ label-exp ⟩ → ‘{’ ⟨ field-exp ⟩ (‘,’ ⟨ field-exp ⟩)* ‘}’
⟨ field-exp ⟩ → ⟨ index ⟩ ‘=’ ⟨ term ⟩ | ⟨ composition ⟩ | ⟨ rest ⟩
⟨ rest ⟩ → ‘...’ | ‘-’ | ‘X’ | ‘U’
⟨ composition ⟩ → ⟨ rest ⟩ ‘;’ ⟨ rest ⟩
⟨ cond ⟩ → ⟨ eq ⟩ | ⟨ pred ⟩ | ⟨ inst ⟩ | ⟨ cond-step ⟩
⟨ eq ⟩ → ⟨ term ⟩ ‘=’ ⟨ term ⟩
⟨ pred ⟩ → ⟨ term ⟩
⟨ inst ⟩ → ⟨ term ⟩ ‘:=’ ⟨ term ⟩
⟨ declaration ⟩ → ⟨ transition ⟩

The ⟨ transition ⟩ non-terminal specifies how MSOS transitions are written in MSDF. Let us begin our description with unconditional transitions (⟨ uncond-trans ⟩), described by the non-terminal ⟨ step ⟩, with an optional label ⟨ label ⟩.[2] An unconditional transition establishes three possible relations between terms: "big-step" semantics (‘==>’),

[2]Which is only decorative and should not to be confused with the MSOS label.

"small-step" semantics (`-->`), and static semantics (also `==>`, overloaded), following the traditional notation of operational semantics literature [52, 58, 54]. All three relations, described by the non-terminal ⟨ relation ⟩, are ternary with the following components: the typed value-added syntactic tree to be matched against, the MSOS label expression (⟨ label-exp ⟩) enclosed between braces, and the resulting value-added syntactic tree.

A label expression is a non-empty list of field expressions (⟨ field-exp ⟩) separated by commas and enclosed between braces, where each field expression is either an index and its component or the "rest of the label." There are special metavariables for the rest of the label (⟨ rest ⟩): if it is unobservable, the metavariable '`-`' should be used, otherwise '`...`' should be used. As usual on MSOS specifications, we use `-->` and `==>` as sugar for `-{-}->` and `={-}=>`, respectively. Instead of '`...`' and '`-`', one may uses the metavariables '`X`' and '`U`' respectively. These metavariables may optionally be postfixed with a number, such as '`X1`'.

Label composition is specified using the syntax of the non-terminal ⟨ composition ⟩. It is recommended that numbered metavariables ('`X1`', '`X2`', etc.) be used on the composition to ease the understanding and to follow traditional MSOS notation. Of course, label composition only makes sense on conditional transitions.

Metavariables over sets are not declared explicitly in MSDF, but instead declared implicitly: all non-terminals of the form ⟨ dsetid ⟩ that appear in transitions are considered metavariables for their corresponding sets using this simple formation rule: given that all set identifiers are assumed to have only letters (Section 4.2.2), all characters that are neither letters nor the symbols '`*`' and '`+`' are stripped from the ⟨ dsetid ⟩ and the remaining is assumed to be the intended type of the metavariable.

For example: '`Exp`', '`Exp1`', and '`Exp'`' are all metavariables that range over '`Exp`', obtained by removing the characters '`1`' and '`'`', respectively, while '`Exp*1`' is a metavariable over '`Exp*`', obtained by removing the character '`1`'. The rationale for this is that, in programming language definitions, it is often the case in which a set is used on most transition rules. For example, the set of expressions, or the set of values. As in the case of module inclusions, it would probably be tedious and error prone to declare the same variables over the same modules.

As an example of unconditional transitions, label expressions and implicit metavariables let us consider the following fragment that defines the semantics of a construction that prints a computed value. It defines an unconditional transition between the typed syntactic tree '`(print Value) : Cmd`' to '`skip`', the "do-nothing" command. The semantics of the '`print`' command is defined with a write-only component by the label expression '`out' = Value`' that models the produced information '`Value`'. The rest of the record is *unobservable*, that is, any read-write component remains unchanged, no other produced information occurs on other write-only components, and read-only components will always remain the same, of course.

```
(print Value) : Cmd -{out' = Value, -}-> skip .
```

Conditional transitions have four different types of conditions, ruled by the non-terminal ⟨ cond ⟩: equality conditions, predicates, variable instantiations, and conditional transitions. Equality conditions (⟨ eq ⟩) assert the equality of two terms, such as '`first`

`(Pids') = Int`'. A single term '`P`' is used as a predicate (⟨ `pred` ⟩), such as '`odd(n)`', to abbreviate the equational condition '`P = true`'. The ⟨ `inst` ⟩ non-terminal defines the syntax used to instantiate new metavariables. The free metavariable must be on the left-hand side of the '`:=`', as in '`Value := lookup (Id, Env)`'. Finally, ⟨ `cond-step` ⟩ is a conditional transition that has the same syntax as of unconditional transitions, with the exception that the type of the syntactic tree to be matched against is necessarily the *least set* that applies to the left-hand side term, that is, the smallest set in a set inclusion relation.

Let us exemplify conditional transitions with a small-step specification for the semantics of an abstract conditional construction, with syntax '`cond Exp Exp Exp`'. The semantics is the usual: the first expression is to be evaluated into a boolean value; if it is true, the second expression is evaluated, otherwise the third is. The semantics for this needs three transitions: the first states that the condition—the metavariable '`Exp`' ranging over the set '`Exp`'—must be evaluated. The term '`(cond Exp Exp1 Exp2) : Exp`' is a typed syntactic tree with the explicit type '`Exp`', while on the condition, the typed syntactic tree '`Exp`' has as implicit type the least set applicable to the metavariable, which is also '`Exp`'. The condition states that, if '`Exp`' is evaluated into '`Exp'`', then the left-hand side of the main transition evaluates to '`cond Exp' Exp1 Exp`'. The label '`...`' on the condition is the same as the main transition, meaning that any changes to read-write components are propagated to the main transition, and any produced information by write-only components is also produced by the main transition. Read-only components must always remain the same.

```
                       Exp -{...}-> Exp'
-- -----------------------------------------------------
(cond Exp Exp1 Exp2) : Exp -{...}-> cond Exp' Exp1 Exp2 .
```

In the example above, we used the fact that, in Maude, three dashes ('`---`') initiate a comment line.

Finally, two additional rules are needed for each possible outcome of '`Exp`': if it is '`tt`', the whole left-hand side evaluates to '`Exp1`', otherwise into '`Exp2`'. These transitions are unobservable.

```
(cond tt Exp1 Exp2) : Exp --> Exp1 .
(cond ff Exp1 Exp2) : Exp --> Exp2 .
```

Let us further exemplify conditional transitions by defining the same conditional construction in a big-step style, which also uses three rules, but with an additional construction, as follows:

The internal construction (not part of the main syntax of the language) '`if-choose`' has three arguments: one value and two expressions. If the value is '`tt`', then it evaluates to '`Exp2`', otherwise to '`Exp3`'. The example also shows each transition rule with a rule label.

```
Exp ::= if-choose (Value, Exp2, Exp3) .
```

```
[if-choose-tt] if-choose (tt, Exp2, Exp3) : Exp ==> Exp2 .
[if-choose-ff] if-choose (ff, Exp2, Exp3) : Exp ==> Exp3 .
```

There is a single transition for the conditional construction itself, but with three conditional transitions. The first expects the expression '`Exp`' to be evaluated into '`Value`', with label expression '`X1`'. This value, when used as a parameter in '`if-choose (Value, Exp2, Exp3)`', is expected to be evaluated, in turn, into '`Exp'`', in an unobservable manner. Finally, '`Exp'`' is expected to be evaluated into '`Value'`', with label expression '`X2`'. If those three conditions are satisfied and label expressions '`X1`' and '`X2`' are composable—recalling Section 2.1, the read-only components remain the same, the initial value of a read-write component on '`X2`' is the final value of the same read-write component on '`X1`', while produced information does not affect composability of labels—, then the whole left-hand side is evaluated into '`Value'`'. The label '`X1 ; X2`' on the main transition specifies that the resulting label is the composition of labels '`X1`' and '`X2`'.

```
      Exp ={X1}=> Value,
      if-choose (Value, Exp1, Exp2) ==> Exp',
      Exp' ={X2}=> Value'
 [if] -- ----------------------------------------------
      (cond Exp Exp1 Exp2) : Exp ={X1 ; X2}=> Value' .
```

MMT checks for source-dependent variables (see Section 2.1) and reports when it finds transitions that contain variables without this property, as the following example shows:

```
(msos TEST is
 Foo .
 Bar .

 Foo ::= f (Bar, Bar) .

         Bar --> Bar'
 -- -------------------------
 f(Bar, Bar) : Foo --> Bar'' .
sosm)
```

Upon reading module '`TEST`', the following error is displayed:

```
rewrites: 19081 in 96ms cpu (101ms real) (196741 rewrites/second)
ERROR: non source-dependent variables found: Bar'' in module TEST
```

4.3 Built-in operations on derived and parameterized sets

The parameterized and derived sets in MSDF have several associated operations, which we describe in this Section. Each function is presented as $\mathsf{f} : \mathsf{S} \to \mathsf{S}'$: a function f with

domain S and codomain S′. If the function f is in mixfix format, it is converted to prefix format, where the arguments are replaced by underscores (‘_’).

Let s, and s′ be any two primitive sets (that is, neither derived nor parameterized sets). Recall that, from a set s, the sets s* and s+ are automatically derived. Furthermore, the user may create the following parameterized sets: (s)List, (s)Set, and (s, s′)Map.

4.3.1 Sequences

_,_ : s* × s* → s*

The monoid binary function (with optional surrounding parentheses), with identity is ‘()’. A single element s is also a sequence.

4.3.2 Lists

[_] : s* → (s)List

Constructs a list out of a sequence of elements.

_in_ : s × (s)List → Bool

Returns *true* if the s is on the list (s)List.

first : (s)List → s

Returns the first element from the list (s)List.

remove : s × (s)List → (s)List

Creates a copy of the list (s)List with all copies of s removed.

insert-back : s × (s)List → (s)List

Inserts s as the last element of (s)List.

insert-front : s × (s)List → (s)List

Inserts s as the first element of (s)List.

length : (s)List → Nat

Number of elements of (s)List.

4.3.3 Maps

_|->_ : s × s′ → (s,s′)Map

Creates an entry that binds s to s′.

_+++_ : (s,s′)Map × (s,s′)Map → (s,s′)Map

Disjoint union of maps.

length : (s,s′)Map → Nat

Number of entries on the map.

`def lookup` : `s` × `(s,s′)Map` → `Bool`

Is *true* if there exists a mapping from `s` in `(s,s′)Map`.

`lookup` : `s` × `(s,s′)Map` → `s′`

Returns the element that `s` maps to in `(s,s′)Map`.

`_/_` : `(s,s′)Map` × `(s,s′)Map` → `(s,s′)Map`

Overrides the mappings of the second `(s,s′)Map` with the ones of the first `(s,s′)Map`.

4.3.4 Sets

`size` : `(s)Set` → `Nat`

Number of elements of `(s)Set`.

`_in_` : `s` × `(s)Set` → `Bool`

Is *true* if `(s)Set` contains `s`.

`_+_` : `(s)Set` × `(s)Set` → `(s)Set`

Disjoint union of sets.

4.4 User interface

The normal operation of the *Maude MSOS Tool* is with the user inputting MSDF modules at the command prompt or by loading files using Maude's '`load`' command and by using Full Maude's own commands for rewriting, reducing, searching and model checking specifications. This Section outlines the commands that are specific to *Maude MSOS Tool*.

Currently the only command available controls the *step flag* of the compilation process (Chapter 5 details the compilation process). The normal operation of the MMT is with this flag *on*, since, as shown in Section 2.4, there is a need to control the rewrites on the conditions. The only case in which this step must be *off* is when there is other means of controlling these rewrites, for example, using Alberto Verdejo's strategy language for Maude [45]. The syntax of the command is either '`step flag on`' or '`step flag off`'.

4.5 A simple example

This section revisits the simple example of a language specification first shown in Section 2.4, adapted for*Maude MSOS Tool*. Like the example on Section 2.4, this very simple programming language contains only an ML-like '`let-in-end`' construction and the '`sum`' function.

The following specification is enclosed by an MSOS module definition to be accepted in the MMT.

```
(msos SIMPLE-LANGUAGE is ... sosm)
```

The following datatype definitions declares the sets used on the specification: '`Exp`', the set of expressions, and '`Id`', the set of identifiers. We let the expressions range over identifiers and integers (the built-in set '`Int`'), the only primitive type of our simple programming language.

```
Id .
Exp .
Exp ::= Int | Id .
```

We now declare the two constructions of the language.

```
Exp ::= let Id = Int in Exp end
      | Exp sum Exp .
```

The specification of the '`let-in-end`' construction requires a read-only environment for the bindings. We declare it as a map from identifiers to integers. This environment is accessed by the index '`env`'.

```
Env = (Id, Int)Map .
Label = { env : Env, ... } .
```

Now for the dynamic semantics. To evaluate the sum of two expressions, we first evaluate the first expression until it reaches a final value, which is specified to be an integer in this language. Then the second expression is evaluated. When final values are produced, the final value of the function itself is the mathematical sum of the two integers.

```
                 Exp1 -{...}-> Exp'1
-- -----------------------------------------
(Exp1 sum Exp2) : Exp -{...}-> Exp'1 sum Exp2 .

                 Exp2 -{...}-> Exp'2
-- ---------------------------------------
 (Int sum Exp2) : Exp -{...}-> Int sum Exp'2 .

       Int3 := Int1 + Int2
-- ----------------------------
(Int1 sum Int2) : Exp --> Int3 .
```

For the meaning of the '`let-in-end`' construction, the expression is evaluated in the context of the current environment overridden with the binding provided by '`Id = Int`' declaration. When the evaluation of the expression reaches the final value of an integer, the whole expression evaluates to this integer.

```
   Env' := (Id |-> Int) / Env, Exp -{env = Env', ...}-> Exp'
 -- -------------------------------------------------------
 (let Id = Int in Exp end) : Exp -{env = Env, ...}->
                                 (let Id = Int in Exp' end) .

 (let Id = Int in Int' end) : Exp --> Int' .
```

The evaluation of an identifier looks up its mapping in the environment and returns it.

```
    Int := lookup (Id, Env)
 -- -------------------------
 Id : Exp -{env = Env, -}-> Int .
```

In order to run programs with our specification we need to provide identifiers to it. We do this by creating constants '`x`' and '`y`'.

```
(msos TEST is
 see SIMPLE-LANGUAGE .

 Id ::= x | y .
sosm)
```

We may now use Full Maude's '`rewrite`' command to execute a simple program, whose argument is an MRS configuration (see Section 2.4).

```
(rewrite < (let x = 10 in
             let y = 10 in
              x sum y
             end
            end) ::: 'Exp,
            { env = void } > .)

rewrite in TEST : < ... >
result Conf :
  < 20 ::: 'Exp, { env = void } >

Bye.
```

Chapter 5

The implementation of MMT

This Chapter describes the implementation of *Maude MSOS Tool*. Recall from Section 2.3.2.1 that the representation of a language/logic $\mathcal{L}$ in rewriting logic is given by a representation map $\Psi : \mathcal{L} \to \mathcal{R}$ which is ultimately expressed by an executable function in Maude $\overline{\Phi}$: $\texttt{Module}_{\mathcal{L}} \to$ `Module`. In the case of MSOS, this function is $\overline{\Phi}$: $\texttt{Module}_{\mathcal{M}} \to$ `Module`, where $\texttt{Module}_{\mathcal{M}}$ is a data type representing MSDF modules and, as usual, `Module` is the data type representation of Full Maude's system modules.

This Chapter is organized as follows: Section 5.1 describes procedure first described in Section 2.3.4.3 applied to the case of MSDF modules; Section 5.2 describes the highest-level of the compilation process, that of *MSDF modules*; Section 5.3 describes how datatype specification in MSDF is converted to Maude; Section 5.4 describes how label information is used on the compilation process; and finally, Section 5.5 describes how MSDF transitions are compiled into rewriting logic rules.

5.1 MMT as an extension of Full Maude

Since MMT is meant to be a conservative extension of Full Maude, we need to parse both MSDF and Full Maude input. That is, we need not only to create the signature for the MSDF constructions, but also to *combine* that with the Full Maude signature to parse user input. The functional module '`MSOS-SL-SIGNATURE`' defines the signature of MSDF declarations and we combine with Full Maude's '`GRAMMAR`' to create the module '`MSOS+FM-GRAMMAR`'. This combined signature is used on the command loop rules to parse user input and redirect to the correct handlers. For this, we need to replace the '`FULL-MAUDE`' module with our own, named '`MAUDE-MSOS-TOOL`', that contains such rules. This module includes '`EXT-DATABASE-HANDLING`' which extends the '`DATABASE-HANDLING`' with the functions that apply to MSDF user input.

Let us give a concrete example. Recall from Section 2.3.4.3 how Full Maude user input is processed. In the particular case of MMT, when the user types a text through '`LOOP-MODE`', it is converted to '`QidList`' and matched on the rule '`[in]`' on the variable '`QIL`'. The rule then attempts to parse this token list with the combined signature and, if it is successful, puts the parsed term on the '`input`' attribute of the database object.

```
crl [in] : [QIL, < O : X@Database |
```

```
                 db : DB, input : nilTermList, output : nil,
                 default : MN, Atts >, QIL'] =>
         [nil, < O : X@Database | db : DB,
                input : getTerm(metaParse(MSOS+FM-GRAMMAR,
                                QIL, 'Input)),
                output : nil, default : MN, Atts >, QIL']
if QIL =/= nil /\
   metaParse(MSOS+FM-GRAMMAR, QIL, 'Input) : ResultPair .
```

The '`EXT-DATABASE-HANDLING`' module declares specific rules for each type of input term. Let us show one example. The following rule works as follows: if the input term matches against the pattern '`'msos_is_sosm[T, T']`', which is an MSDF module as defined by the module '`MSOS-SL-SIGNATURE`', it is given to the function '`mmt-proc-unit`' that begins the compilation process.

```
rl < O : X@Database | db : DB, input : ('msos_is_sosm[T, T']),
                      step-flag : B,
                      output : nil, default : MN, Atts > =>
   < O : X@Database | db : mmt-proc-unit('msos_is_sosm[T, T'],
                      step-flag(B), DB),
                      input : nilTermList, step-flag : B,
                      output : ('Introduced 'MSOS 'module
                             modNameToQid(parseModName(T)) '\n),
                      default : parseModName(T), Atts > .
```

The entire sequence works as follows: the metaparsing of the user entered '`QidList`' produces a term of sort '`MSOSUserInput`' that is the metarepresentation of the user input, defined by '`MSOS-SL-SIGNATURE`' module. We need to move from this metarepresentation to a term of sort '`MSOSModule`' that is the metarepresentation of MSDF modules, as defined on the '`MSOS-UNIT`' module. This two-phase process—user input into '`MSOSUserInput`' and then to '`MSOSModule`'—is used because an MSDF module has *user-definable syntax*, where the terms that appear in transitions are usually defined on the *same* module. Because of this at the time of the first phase there is no information about the user defined signature of these. The solution adopted by the Maude system and followed by MMT is to use *bubbles* in place of those terms. Bubbles are sequence of tokens that have not been parsed yet because there is no signature to parse them. After the signature is collected from the MSDF module, it is used to parse those bubbles. Once all the bubbles have been parsed, we have a *complete* metarepresentation of MSDF modules, according to the definition of the '`MSOS-UNIT`' module and the compilation may begin.

The first phase of the parsing produces a term that is the metarepresentation of a '`MSOSUserInput`' module. This is not the MSDF module, as we have said, but a metarepresentation of what the user has typed. From this module we obtain the '`MSOSModule`', constructing it with the '`mmt-proc-unit`', defined on the module '`MSOS-PARSER`', with the following signature:

$$\texttt{mmt-proc-unit} : \texttt{Term} \times \texttt{CFlags} \times \texttt{Database} \rightarrow \texttt{Database}$$

where '`Term`' is the parsed term, '`CFlags`' are the *compilation flags* that control the compilation process, described in Section 5.2, and '`Database`' is the database of modules. In order words, this function receives a term and a database, and returns a database modified by inserting the "term unit" of the MSDF module; recall from Section 2.3.4 that the "term unit" is the metarepresentation of the user input.

After inserting the term unit into the database, '`mmt-proc-unit`' gives control to '`mmt-eval-preunit`', whose job is to construct a signature out of the datatype information from the MSDF module and solve the bubbles in it. The function is defined on the module '`MSOS-SOLVER`' and has the following signature:

$$\texttt{mmt-eval-preunit} : \texttt{Unit} \times \texttt{CFlagsDatabase} \rightarrow \texttt{Database}$$

This functions expects to receive a '`Unit`', which is a supersort of the '`MSOSModule`' sort. It then solves the bubbles on this module by calling the function '`solve-module-bubbles`' from module '`MSOS-SOLVE-BUBBLES`'.

$$\texttt{solve-module-bubbles} : \texttt{Unit} \times \texttt{Database} \rightarrow \texttt{Unit}$$

After the bubbles have been solved, the control returns to the '`mmt-eval-preunit`' function, which then calls '`convertMSOS`', which is the function that actually produces the rewrite theory associated with the MSDF specification. The rest of this Chapter is dedicated to the functionality provided by this function.

5.2 Modules

Let us begin by giving a high level view of the compilation process. Given an MSDF specification $\mathcal{M}$, each *MSOS module* is converted into a *system module* that uses the MRS rewrite theories; datatype definitions from $\mathcal{M}$ are converted into axioms in membership equational logic, and the three-element relations (big-step, small-step, and static semantics) are converted into the two-element relation defined by rewriting logic rules over MRS configurations; *label expressions* are converted into MRS *record expressions*, and must be split into two parts: records at the left-hand side (*before*) and at the right-hand side (*after*) of MRS rules.

In order to formalize this, let us consider an abstract MSDF module $\mathcal{M}$ as a tuple $(\mathsf{D}, \mathsf{L}, \mathsf{T})$, with D the datatype declarations, L the label declarations, and T the transition rules. The compilation process generates a Full Maude system module `Module` that contains the signature $(\Sigma, \mathsf{E}, \mathsf{R})$, with Σ the signature, E the set of equations, and R the set of rewrite rules. The signature Σ contains the poset $(\mathsf{S}, \subseteq)$ of sorts, and the set O of operators.

In the MMT this high level operation is performed by the function '`convertMSOS`', defined on the module '`MSOS-CONVERTER`', with the following signature:

$$\texttt{convertMSOS} : \texttt{MSOSModule} \times \texttt{CFlags} \times \texttt{Database} \rightarrow \texttt{StrSModule}$$

where 'MSOSModule' is the actual data type of Module$_{\mathcal{M}}$ in the MMT implementation. The image of 'convertMSOS' is a 'StrSModule', the metarepresentation, in Full Maude, of Module modules. The sort 'CFlags' represents the flags that control the compilation process; currently only one flag is used, the *step* flag that controls the generation of *restricted configurations* that contains the 'step' rule, and the operators '{_,_}' and '[_,_]', see Section 5.5. The sort 'Database' is the database of modules handled by Full Maude.

5.3 Datatypes

Now for the details of the compilation process, let us begin with the description on how datatype information is processed. It is known since the early development of many-sorted equational logic that there is a correspondence between many-sorted equational logic and context-free grammars [24] and this fact is explored on the conversion from MSDF's datatype information into Maude signatures. However, we postpone the inclusion of derived and implicit datatypes (such as lists, sequences, etc.) to Section 5.3.5.

Informally, the datatype declarations in D are used to generate the signature Σ, by converting each set declaration into a sort declaration in S, each subset inclusion into a subsort inclusion, and each function declaration into an operator in O.

5.3.1 Compilation of type declarations

The contents of the restricted configuration module ('RCONF', see Section 2.4) are also created as a many-sorted signature from D, and, in order to understand why this is needed, let us recall from Section 2.4 the definition of MRS configurations and restricted configurations: MRS has three configuration constructors, '<_,_>', '{_,_}, and '[_,_]', with a single step rule; the operators and the rule range over the abstract sort 'Program' and the sort 'Record'. Having one sort for the program text requires that *every* sort that appears on the first projection of configurations must be a subsort of 'Program'. While this simplifies the number of elements on a Maude system module, it also has the drawback of putting all sorts, even those originally under separated components, into the same connected component, which creates restrictions of regularity and overloading rules that are artificially introduced by the compilation process and not by the user—a source of frustration. To avoid some of these problems, the compilation process creates the three operators for each sort. The same argument is applied to the 'Component' sort, the '_:_' constructor for fields, and the 'step' rule. Because of this, a good deal of the compilation process involves the *reconstruction* of some of the elements present of those modules that are now absent due to the removal of the 'Program' and 'Component' sorts.

Formally speaking, consider the datatype definition $\mathsf{D} = (\mathsf{S}_\mathrm{D}, \mathsf{F}_\mathrm{D})$ where S_D is the poset of sets and F_D is the set of pairs of function signatures and a set of attributes. Then, for each new set $\mathsf{s} \in \mathsf{S}_\mathrm{D}$, a sort $\mathsf{s} \in \mathsf{S}$, the poset of sorts in a rewrite theory signature Σ, is created, with the same name. Each subset inclusion defines the order on the poset S: for each $\mathsf{s} \subseteq \mathsf{s}'$, with $\mathsf{s}, \mathsf{s}' \in \mathsf{S}_\mathrm{D}$, its corresponding sorts are related as $\mathsf{s} < \mathsf{s}'$ in Σ.

This is implemented by functions '`get-new-sorts`' and '`get-subsorts`', defined on module '`BNF-TOOLS`'. They have the following syntax:

$$\begin{aligned} \texttt{get-new-sorts} &: \texttt{Set<BNF>} \to \texttt{Set<ESort>} \\ \texttt{get-subsorts} &: \texttt{Set<BNF>} \to \texttt{Set<ESubsortDecl>} \end{aligned}$$

where '`Set<BNF>`' is the metarepresentation of the datatype declarations on an MSDF module, '`Set<ESort>`' is the metarepresentation of sort declarations, and '`Set<ESubsortDecl>`' is the metarepresentation of subsort declarations.

Given a poset $(S, \subseteq)$ of sort declarations, let S_{max} be the set of *maximal* supersorts in S. That is, S_{max} contains only the top sorts on each connected component of S. For each sort $s \in S_{max}$, the $\texttt{<_,_>} : s \times \texttt{Record} \to \texttt{Conf}$ operator is declared in Σ. S_{max} is used instead of S to avoid adding unnecessary rules and operators to the theory, since all operators and rules that apply to a sort also apply to all of its subsorts.

If the step flag is *on*, then the following additional operators are also created:

$$\begin{aligned} \{_,_\} &: s \times \texttt{Record} \rightharpoonup \texttt{Conf} \\ [_,_] &: s \times \texttt{Record} \rightharpoonup \texttt{Conf} \end{aligned}$$

This functionality is implemented by the function '`make-confs`' defined on the module '`AUX-CONF-OPS`', with the following signature:

$$\texttt{make-confs} : \texttt{Set<BNF>} \times \texttt{CFlags} \to \texttt{Set<EOpDecl>}$$

where '`Set<EOpDecl>`' is the metarepresentation of operators in a Full Maude system module.

5.3.2 Compilation of typed syntactic trees

A typed syntactic tree used in transitions is represented by a pair '$\texttt{t ::: q}$', where $\texttt{t}$ is the term representing the syntactic tree and $\texttt{q}$ is a quoted-identifier that represents the sort. This quoted-identifier is the name of the sort prefixed with a single quote, such as '`'Exp`'. So, for each sort $s \in S_{max}$, the following operator is declared in Σ:

$$_:::_ : s \times \texttt{Qid} \to s$$

This compilation step is implemented by the function '`make-tst-ops`' defined on the module '`AUX-TYPED-SYNTACTIC-TREE-OPS`'. It has the following signature:

$$\texttt{make-tst-ops} : \texttt{Set<BNF>} \to \texttt{Set<EOpDecl>}$$

5.3.3 Compilation of functions

Continuing, for each function declaration $f : s_1 \times \cdots \times s_n \to s$ in F_D the following operator is added to O as follows:

$$f : s_1 \times \cdots \times s_n \to s$$

Any attribute in f is moved verbatim to the operator. The function f may be in mixfix format. In this case, the operator name is constructed by keeping all lowercase identifiers from f and substituting all uppercase identifiers by underscores (‘_’) and the domain of the operator is the set of all uppercase identifiers in f.

The conversion from functions to operators is done by the function ‘bnf->ops’, defined in the module ‘BNF-TOOLS’. It has the following signature:

$$\texttt{bnf->ops} : \texttt{Set<BNF>} \to \texttt{Set<EOpDecl>}$$

Next, the step rules from MRS must be generated. For search sort $s \in S_{max}$, the following rule is generated if the step flag is *on*.

$$\begin{array}{ll} \textbf{crl} & \texttt{<}X ::: qid(s), R\texttt{>} \to \texttt{<}X' ::: qid(s), R'\texttt{>} \\ \textbf{if} & \{X ::: qid(s), R\} \to [X' ::: qid(s), R'] \quad [step] \end{array}$$

where X, X' are variables of the sort s, and R, R' are variables of the sort ‘Record’. The function $qid(s)$ converts a sort name into a quoted-identifier. If the step flag is *off*, no step rule is generated. The one-step rewrite at the conditions, in this case, must be controlled by some other means, like a rewriting strategy.

5.3.4 Compilation of module inclusion

Finally, we must deal with module inclusion. The modules ‘QID’ and ‘MSOS-RUNTIME’ are always included by the rewrite theory being generated. The former is needed due to the use of quoted-identifiers on typed syntactic trees representations in Maude (the ‘_:::_’ operator), and the later contains the basic runtime support of the execution of MSDF modules, such as the definition of the ‘Record’ sort, the sorts of indices, etc. For each module m included using syntax ‘see m’, a corresponding ‘including m’ is generated. Each parameterized set declaration gives rise to the inclusion of a module expression of the relevant parameterized module. The details of this are described in Section 5.3.5 and in Table 5.1.

For the implicit module inclusions described in Section 4.2.1, we need a few definitions first: let $sets(D)$ be a projection function that returns all set identifiers that are mentioned on an MSDF module datatype definition D, $newsets(D)$ a projection function that returns all set identifiers that correspond to the *new* sets declared on D, and $modules(\{s_0, \ldots, s_n\})$ returns the modules that declare the sets identifiers in the set $\{s_0, \ldots, s_n\}$. Then the list of implicit modules to be included is $modules(sets(D) \setminus newsets(D))$, which is the list of the modules that declare the sets that are not new in

the current set.

This functionality is implemented by the function 'make-includes', defined in the module 'MSOS-INCLUDE-GENERATION', with the following signature:

make-includes : MSOSModule × CFlags × Database → List<EImport>

where 'List<EImport>' is the metarepresentation of the list of imports in a Full Maude module.

This function, in turn, depends on the following functions. The function 'see->import' converts MSDF inclusions into Maude inclusions, with the following signature:

see->import : List<See> × Database → List<EImport>

where 'List<See>' is the metarepresentation of the list of imports in an MSDF module.

The function 'generate-subsort-includes' handles implicit module inclusions:

generate-subsort-includes : List<See> × Set<BNF> ×
LabelType × Database → List<EImport>

where 'LabelType' is the metarepresentation of label declarations.

Finally, 'generate-parameterized-includes' handles the inclusion of the parameterized modules.

generate-parameterized-includes : Set<BNF> × Database
→ List<EImport>

These functions are all declared on the same module 'MSOS-INCLUDE-GENERATION'.

The function that implements the search for a declaring module is 'find-declaring-module', defined on the module 'SORT-SEARCH-TOOLS'. It has the following signature:

find-declaring-module : Set<ESort> × Database → Set<ModName>

The sort 'Set<ModName>' is a set of module names. Function 'find-declaring-module' receives a set of sorts and returns the set of modules where they are defined. Currently, the tool expects that there exists a single module that defines each sort. This function works by iterating through all modules in the Full Maude database, extracting the list of sorts that are defined on each module, and checking to see if the one of the given sorts is present on this list. (It carefully avoids checking the internally generated abstract modules A(s) described in Section 5.3.5 that are used to

Type	*Module*	*Sorts*
Sequences	`SEQUENCE(X :: TRIV)`	`Seq(X)`, `NeSeq(X)`
Sets	`SET(X :: TRIV)`	`Set(X)`
Lists	`LIST(X :: TRIV)`	`List(X)`
Maps	`MAP(X :: TRIV | Y :: TRIV)`	`Map(X | Y)`

Table 5.1: Relationship between parameterized types and Full Maude

resolve the view forwarding problem.)

To exemplify the compilation so far, consider the following MSDF fragment that assumes that the set '`Value`' is defined in the module '`VALUE`'. This fragment defines two new sets, the abstract syntax of two constructions, a looping command, and a parallel execution command, two subset inclusions, and has an explicit inclusion of another MSDF module.

```
see SEQ .

Id . Exp .
Exp ::= Id
      | Value
      | while Exp do Exp
      | parallel (Exp, Exp) .
```

With the step flag *on*, it is converted into the following Maude fragment:

```
include VALUE .            op <_,_> : Exp Record -> Conf .
include SEQ .              op {_,_} : Exp Record ~> Conf .
                           op [_,_] : Exp Record ~> Conf .
sort Id .                  op _:::_ : Exp Qid -> Exp .
sort Exp .
subsort Id < Exp .         op while_do_ : Exp Exp -> Exp .
subsort Value < Exp .      op parallel : Exp Exp -> Exp .
```

5.3.5 Parameterized and derived types

Let us now extend the process so far to derived and parameterized types. Since the translation involve a large number of steps, the simplest ones are presented first, enlarging the conversion incrementally, while justifying the need for each increase in complexity.

Parameterized sets in MSDF—lists, sets, and maps—are converted into *parameterized sorts*, a feature only available in Full Maude in the version 2.1.1 of the Maude system. For each parameterized type, there is a built-in parameterized Full Maude system module (see Section 2.3 for a discussion on those) that implements the generic functionality of the relevant type. Table 5.1 lists the relationship between MSDF's parameterized types and Full Maude's parameterized modules and sorts. Recall that in order to instantiate a

parameterized module with a specific sort, we first need to create a view; then, module instantiation is made by importing a module expression that involves the parameterized module and the desired view (see Section 2.3.4). So, for each parameterized instantiation in an MSDF module the corresponding module expression that instantiates the parameterized module with the view is added to the includes section of the generated Full Maude system module. This module instantiation makes available the parameterized sorts that corresponds to the parameterized types, according to Table 5.1. Recall from Section 4.2.2 that parameterized sets are not used directly by themselves, but through an *equivalent set*—the left hand side of the ⟨ equiv ⟩ non-terminal. This equivalent set of the parameterized set is made a supersort of the parameterized sort. This is necessary so that all elements of the parameterized sort algebra will be made available to the equivalent sort algebra, including their carrier sets and operations.

In order to formalize this, let us consider an abstract declaration of an equivalent set s with its parameterized set $\mathsf{P}(\mathsf{s}_0, \ldots, \mathsf{s}_n)$ as $\mathsf{s} = \mathsf{P}(\mathsf{s}_0, \ldots, \mathsf{s}_n)$ in an MSDF module $\mathcal{M}$. This declaration gives rise to the following components in the generated rewrite theory: a sort declaration for s in Σ; an include for the parameterized module instantiation, which depends on P, according to Table 5.1, as the instantiated view depends on $\mathsf{s}_0, \ldots, \mathsf{s}_n$; and, finally, a subsort ordering that relates s and $\mathsf{P}(\mathsf{s}')$ as $\mathsf{P}(\mathsf{s}') < \mathsf{s}$ is added to S.

For example, the declaration '`Channels = (Channel)Set`' generates the following Maude fragment, where the module expression '`SET(Channel)`' is the instantiation of the parameterized module '`CHANNEL`' with the view '`Channel`' (not shown here). The equivalent set '`Channels`' is converted into the sort '`Channels`' and is made a supersort of the parameterized sort '`Channel(Set)`'.

```
mod CHANNEL is
 including SET(Channel) .

 sort Channels .
 subsort Channel(Set) < Channels .
endm
```

5.3.5.1 View forwarding problem

The flexibility of MSDF creates a problem with this scheme that manifests itself when a parameterized type is instantiated on the same module that its parameter set is declared, such as:

```
msos CHANNEL is
 Id .
 Env = (Id, Int)Map .
sosm
```

The problem is that a view must be defined *before* a module instantiation occurs—recall that a view defines a mapping from one theory (usually '`TRIV`') to a module. In this case, the view cannot be defined because the sort it refers to is defined in the *same* module in which the view is needed. Hence, we have a forwarding declaration problem.

This is solved by adopting the following scheme: for each sort $s \in S$, two *internal modules* are generated and inserted into Full Maude's database before compilation begins: an internal functional module $A(s)$ (for "abstract module"), which contains only the declaration of the sort s and can be seen as an abstract sort declaration module; a view s, that maps the theory 'TRIV' to the module $A(s)$; the view itself maps the sort 'Elt' to the sort s.

After this step is completed, the module instantiation may be used on any subsequent module without any forwarding concerns. The function that creates the abstract modules and views is defined on the 'MSOS-SOLVER' module and has the following signature:

$$\texttt{insert-generated-modules\&views} : \texttt{Set<ESort>} \times \texttt{Database} \rightarrow \texttt{Database}$$

5.3.5.2 Derived Sets

The derived sets treatment on the MMT is similar to the parameterized sets: each derived set is converted into a parameterized sort according to Table 5.1. The difference in the case of derived types is that there is no new sort declaration or any subsort ordering. All references to 's*' and 's+' in the transitions, label declarations, and datatype definitions are automatically converted to 'Set(s)' and 'NeSeq(s)', respectively, during the compilation phase. This is straightforward, since derived types are always defined for any sort s. We could use this same replacement approach for parameterized types (which would avoid some preregularity problems) but this would involve keeping track of all mappings across all defined modules, since a mapping may be defined on a completely separated module from where it is actually used.

Formally, for each sort $s \in S$ in the signature of the generated rewrite theory, the parameterized instantiation of the module expression 'SEQUENCE(s)' is automatically added to its declaring module. This module expression defines 'Seq(s)' and 'NeSeq(s)' that corresponds, respectively, to 's*' and 's+'. In order to maintain consistency among sequences over a lattice of sorts, each subset inclusion $s \subseteq s'$ must generate not only the subsort declaration $s < s'$, but also the subsort declaration of its derived types: $\texttt{Seq(s)} < \texttt{Seq(s')}$, and $\texttt{NeSeq(s)} < \texttt{NeSeq(s')}$. The reason for this is that any sequence of elements from s is also a sequence of elements from s′—consider $(5, 4, 3, 2, 1)$, a sequence of complex numbers, which is also a sequence of reals, integers, and naturals. Since the relation $\texttt{NeSeq(s)} < \texttt{Seq(s)}$ is already built-in on the 'SEQUENCE' module, there is no need to relate NeSeq(s) and Seq(s′), as this is already achieved by transitivity of the $<$ relation.

To exemplify the use of abstract modules and views, consider the following fragment. It defines a set 'Cmd' of, say, commands and creates a function 'seq' that has as a single parameter the set of non-empty list of commands.

```
msos SEQ-CMD is
 Cmd .
 Cmd ::= seq Cmd+ .
sosm
```

The following Maude code is generated (including the generated modules). First, the

abstract module '@@ABSTRACT-Cmd' is generated containing only the declaration of the sort 'Cmd'.

```
fmod @@ABSTRACT-Cmd is
 sort Cmd .
endfm
```

Next the view 'Cmd' is generated that maps 'Elt', defined in 'TRIV', to 'Cmd', defined in '@@ABSTRACT-Cmd'.

```
view Cmd from TRIV to @@ABSTRACT-Cmd is
 sort Elt to Cmd .
endv
```

Then, the module 'SEQ-CMD' is introduced, already including the module expression 'SEQUENCE(Cmd)', which instantiates the parameterized module 'SEQUENCE' with the view 'Cmd'. This instantiation creates the sorts 'Seq(Cmd)' and 'NeSeq(Cmd)'. The function 'seq' is converted into the operator 'seq' and, as described, the set 'Cmd+' is converted directly into the sort 'NeSeq(Cmd)'.

```
mod SEQ-CMD is
 including SEQUENCE(Cmd) .
 sort Cmd .
 op seq_ : NeSeq(Cmd) -> Cmd .
endm
```

In order to exemplify the subsorting of derived types, consider the following module that defines two sets, 'Exp' and 'Id', and makes 'Id' a subset of 'Exp'.

```
msos EXP is
 Id .
 Exp .
 Exp ::= Id .
sosm
```

Omitting the abstract modules and views generated, the following code contains the instantiation of the 'SEQUENCE(Id)' and 'SEQUENCE(Exp)' modules, a subsorting relation between 'Id' and 'Exp', as well as the expected subsorting relation between the derived types.

```
mod EXP is
 including SEQUENCE(Id) .
 including SEQUENCE(Exp) .

 sort Id .
```

```
 sort Exp .
 subsort Id < Exp .
 subsort NeSeq(Id) < NeSeq(Exp) .
 subsort Seq(Id) < Seq(Exp) .
endm
```

The level of nesting in parameterized types is arbitrary. For example, if one needs to create a mapping 'Ref' that maps locations ('Loc') to environments ('Env') which are in turn mapping themselves, one would write:

```
Loc .
Env = (Id, Value) Map .
Ref = (Loc, Env) Map .
```

In order to cope with some preregularity and non-associativity of operators, we currently avoid derived types in parameterized types. Chapter 7 has a discussion on this.

5.4 Processing label declarations

The ⟨label⟩ declaration is used to create the various 'Field' operators that are used on MRS configurations. This is necessary also due to the removal of the 'Component' sort originally used in MRS configurations because of the same problems of preregularity that lead to the removal of the 'Program' sort. Also, a number of additional subsorts of 'Field' and 'Index' are also defined as they are necessary on the compilation of the transition rules. For read-only fields, the sort 'ROField' is used; for read-write fields, the sort 'RWField' is used; and for write-only fields, the sort 'WOField' is used. For the indices, the following sorts are defined: for read-only indices, the sort 'RO-Index' is used; for read-write indices, both sorts 'Pre-RW-Index' and 'Post-RW-Index' are defined related to the unprimed and primed indices, respectively; and finally for write-only indices, the sort 'WO-Index' is used.

Formally, the compilation is as follows. Consider a label declaration L as a set of field declarations $\{\mathsf{f}_0, \ldots, \mathsf{f}_n\}$, where each field f_j is a pair (i, s), with i the index and s the type of the component. Consider a function $sets(\mathsf{L})$ that returns the all the second projections from the fields in L and $indices(\mathsf{L})$ that returns all the first projections from the fields in L. Then, for each set $s \in sets(\mathsf{L})$, its corresponding module $modules(s)$ is included on the module being generated and the following operator is created in O:

$$_\texttt{=}_ : \texttt{Index} \times \texttt{s} \rightharpoonup \texttt{Field}$$

For each index $i \in indices(\mathsf{L})$ the following operator is created in O:

$$i : \to indexsort(i)$$

where $indexsort(i)$ is RO-Index if i is a read-only index, Pre-RW-Index if i is the unprimed index of a read-write index, Post-RW-Index if i is the primed index of a read-write index, and WO-Index if i is a write-only index.

This is implemented by the operator 'make-op-indices', defined in the module 'AUX-INDICES-OPS'. It has the following signature:

$$\texttt{make-op-indices} : \mathsf{LabelType} \longrightarrow \texttt{Set<EOpDecl>}$$

We also need to reconstruct the membership in the original MRS configuration that asserts that the term $\mathsf{i} = \mathsf{s}$ is a field. For each field $(\mathsf{i}, \mathsf{s}) \in \mathsf{L}$ the following membership equation is created:

$$\textbf{mb} \quad (\mathsf{i} = \mathsf{X}) : \mathsf{fieldsort}(\mathsf{i})$$

where X is a variable of the sort s, and $\mathsf{fieldsort}(\mathsf{i})$ is ROField if i is a read-only index, RWField if i is either an unprimed or a primed index of a read-write component, and WOField if i is a write-only index. This is implemented by the operator 'make-memberships', defined in the module 'MSOS-MEMBERSHIP-GENERATION'. It has the following signature:

$$\texttt{make-memberships} : \mathsf{LabelType} \longrightarrow \texttt{Set<EMembAx>}$$

where 'Set<EMembAx>' is the metarepresentation of the set of membership equations in Full Maude.

Recall from that 'duplicate' equations are used on the conditional membership equation that asserts that a record contains no duplicated fields.

$$\texttt{duplicated} : \mathsf{PreRecord} \rightharpoonup \mathsf{Bool}$$

Since there is no single 'Component' sort we have to generate this equation for every sort that appears on a label declaration. Thus, for every sort $\mathsf{s} \in \mathsf{sets}(\mathsf{L})$, the following equation is generated:

$$\textbf{eq} \quad \texttt{duplicated}((\mathsf{i} = \mathsf{X}), (\mathsf{i} = \mathsf{X}'), \mathsf{PR}) = \mathsf{true}.$$

where X, X' are variables of sort s, i is a variable of sort Index, and PR is a variable of sort PreRecord. This is implemented by the operator 'make-dup-function', defined in the module 'AUX-DUP-FUNCTION-EQS', that has the following signature:

$$\texttt{make-dup-function} : \mathsf{LabelType} \longrightarrow \texttt{Set<EEquation>}$$

where 'Set<EEquation>' is the metarepresentation of the set of equations in Full Maude.

As an example, a label declaration such as:

```
Label = { env : Env, st : Store, st' : Store,
          out' : Value*, ... } .
```

generates the following Maude fragment.

```
op _=_ : [Index] [Env] -> [Field] .
```

```
op _=_ : [Index] [Store] -> [Field] .
op _=_ : [Index] [Seq(Value)] -> [Field] .

op env : -> RO-Index .
op out' : -> WO-Index .
op st : -> Pre-RW-Index .
op st' : -> Post-RW-Index .

mb env = V:Env : ROField .
mb out' = V:Seq(Value) : WOField .
mb st = V:Store : RWField .
mb st' = V:Store : RWField .

eq duplicated(I:Index = X:Env,I:Index = X':Env,
              PR:PreRecord) = true .
eq duplicated(I:Index = X:Seq(Value),
              I:Index = X':Seq(Value), PR:PreRecord) = true .
eq duplicated(I:Index = X:Store,I:Index = X':Store,
              PR:PreRecord) = true .
```

5.5 Processing MSOS transitions

This Section presents the implementation of the transformation described in Section 2.4.1. We begin by describing the most straightforward transformation first, that of unconditional rewrites. Essentially, we are converting from a relation between three elements—the two syntactic trees and the MSOS label—to a relation between MRS configurations, which are tuples containing the syntactic tree (or, using the algebraic terminology, the *term*) and the MRS record.

Dealing with the syntactic trees is straightforward: the left-hand side of MSDF transitions become the first projection on the left-hand side configuration, the program part, and the same idea applies to the right-hand side. Recall that, in MSDF, the syntactic trees have an associated *type*; this typed syntactic tree is converted to tuples of syntactic trees and types constructed by the '`_:::_`' operator, as explained in Section 5.3. The use of restricted configurations versus configurations—'`{_,_}`' and '`[_,_]`' versus '`<_,_>`'—is controlled by the step flag.

To handle MSOS label expressions we need to decompose it into two MRS records, one for each MRS configuration: the first represents the fields present in the MSOS label at the *beginning* of the transition, while the second represents the fields present at the *end* of the transition. The decomposition is as follows: read-only fields must remain unchanged, so they appear both at the beginning and at the end with the same value; read-write components are decomposed into their unprimed and primed parts; the unprimed part is moved to the beginning and the primed is moved to the end; finally, write-only fields are more tricky since they model "produced information" and there is no information available at the start of the transition. This is modeled as the *appending* of the produced value to the current sequence that corresponds to that field. Thus, at the end of the

computation, this component will have a sequence of all produced values.

Formally speaking, let us consider the transition as a tuple (c, t, α, t'), where c is the condition, t is the left-hand side, α the MSOS label, and t' the resulting term. In the case of unconditional transitions, c is a tautology. From that, the following unconditional MRS rewrite rule is generated. If the step flag is *on*:

$$\textbf{rl} \quad \{\hat{t} ::: \mathtt{qid}(s), \mathtt{pre}(\hat{\alpha})\} \to [\hat{t'} ::: \mathtt{qid}(s), \mathtt{post}(\hat{\alpha})]$$

If the step flag is *off*, normal MRS configurations are used instead of the restricted configurations above.

$$\textbf{rl} \quad <\hat{t} ::: \mathtt{qid}(s), \mathtt{pre}(\hat{\alpha})> \to <\hat{t'} ::: \mathtt{qid}(s), \mathtt{post}(\hat{\alpha})>$$

Here $\hat{t}$, $\hat{t'}$, and $\hat{\alpha}$ are the same terms as t, t', and α, except that all implicit metavariables have been made explicit according to the rules defined in Section 4.2.4. The expansion of implicit metavariables is a purely syntactical manipulation of terms that is made before any processing is made on the transitions. Given a term t, we remove from it all characters that are neither letters nor the symbols '`*`' and '`+`' to create the sort of that term. This is made by the functions '`create-transition-variables`', '`create-label-variables`', and '`create-condition-variables`', which converts, respectively, terms at the transition, labels and conditions of MSDF transitions.

These functions are implemented on the module '`AUTOMATIC-VARIABLES`' and have the following signature:

$$\begin{aligned} \mathtt{create\text{-}transition\text{-}variables} &: \mathtt{QidList} \to \mathtt{QidList} \\ \mathtt{create\text{-}label\text{-}variables} &: \mathtt{QidList} \to \mathtt{QidList} \\ \mathtt{create\text{-}condition\text{-}variables} &: \mathtt{QidList} \to \mathtt{QidList} \end{aligned}$$

where '`QidList`' is a list of quoted identifiers that represents the input tokens as they are read by '`LOOP-MODE`'.

Before we describe the remainder of the compilation process, a word about how MSOS labels and MRS records are represented algebraically is needed: MSOS labels are defined as purely abstract sorts '`Label`', which represents an entire label; '`ILabel`', which represents an entire, unobservable label; '`FieldSet`', which represents a subset of the fields on a label, and '`IFieldSet`', that represents an unobservable subset of the fields on a label. The equivalent MRS sorts are, respectively, '`Record`', '`PreRecord`', '`IRecord`', and '`IPreRecord`'.

The functions `pre`, `post` convert from *label expressions*—`FieldSet`, `IFieldSet`—in MSOS to *record expressions*—`PreRecord`, `IPreRecord`—in MRS. Let α be a general label expression $\{f_0, \ldots, f_n\}$. Each f_j is a field, which is abstractly represented as pairs (i, c), with i the index and c the component.[1] Let ϵ be an identity field so that $\{f_0, \epsilon, f_1\} = \{f_0, f_1\}$. The conversion rules for complete label expressions are:

[1]Please notice that this is not the same as (i, s), described previously on Section 5.4, which is a label *type declaration*, with s the sort of the component indexed by i.

$$
\begin{aligned}
\mathsf{pre}(\{f_0,\ldots,f_n\}) &= \{\mathsf{pre}(f_0),\ldots,\mathsf{pre}(f_n)\}\\
\mathsf{post}(\{f_0,\ldots,f_n\}) &= \{\mathsf{post}(f_0),\ldots,\mathsf{post}(f_n)\}
\end{aligned}
$$

where the ellipsis are just a compact way of saying that the function ranges over the entire set of fields in a label.

The "rest of the label"-type of variables are converted as follows:

$$
\begin{aligned}
\mathsf{pre}(U) &= \tilde{U}\\
\mathsf{post}(U) &= \tilde{U}\\
\mathsf{pre}(X) &= \tilde{X}\\
\mathsf{post}(X) &= \tilde{X}'
\end{aligned}
$$

where U is a variable of the sort `IFieldSet`, X and X' are variables of the sort `FieldSet`. The result of the conversion generates variables $\tilde{X}$ and $\tilde{X}'$ of the sort `PreRecord`, and $\tilde{U}$, of the sort `IPreRecord`. Since a variable of the sorts '`Label`' or '`ILabel`' is equivalent to the label expression $\{V\}$ with V being a variable of the sort '`FieldSet`' or '`IFieldSet`', there is no need to create equations of the pre and post operations that range over whole labels.

Now let us describe the equations that range over a specific field (i, c). If i is a read-only index:

$$
\begin{aligned}
\mathsf{pre}(i,c) &= (i,c)\\
\mathsf{post}(i,c) &= (i,c)
\end{aligned}
$$

If i is the *unprimed* index of a read-write index:

$$
\begin{aligned}
\mathsf{pre}(i,c) &= (i,c)\\
\mathsf{post}(i,c) &= \epsilon
\end{aligned}
$$

If i' is the *primed* index of a read-write index:

$$
\begin{aligned}
\mathsf{pre}(i',c) &= \epsilon\\
\mathsf{post}(i',c) &= (i',c)
\end{aligned}
$$

If i' is a write-only index:

$$
\begin{aligned}
\mathsf{pre}(i',c) &= (i',V)\\
\mathsf{post}(i',c) &= (i',(V,c))
\end{aligned}
$$

where V is a fresh variable of the sort s. Recall that the only free monoid currently accepted by MMT is the finite sequence, with identity '`()`', and binary operation '`_,_`'.

For conditional transitions, the only type of condition that needs attention is the *transition condition*; the other types are moved verbatim from MSDF to Maude, since they already are present in Maude—they are conditional equations, variable instantiations, and predicates. Conditional transitions are converted into conditional rules, in which transition conditions are converted into rewrite conditions using a slightly different approach from the unconditional transitions. The difference is on the handling of write-only components: at the beginning of a conditional transition the component is the empty sequence '$()$' so that any information produced is present at the conclusion of the transition.

Formally, a conditional transition (c, t, α, t') is converted as follows. Let us consider the condition c a conjunction of conditions $c_0, \ldots, c_n$. If the step flag is *on* the following conditional rewrite rule is generated:

$$\begin{array}{ll} \textbf{crl} & \{t ::: \mathtt{qid}(s), \mathtt{pre}(\alpha)\} \to [t' ::: \mathtt{qid}(s), \mathtt{post}(\alpha)] \\ \textbf{if} & \mathtt{cond}(c_0, \ldots, c_n) \end{array}$$

On the other hand, if the step flag is *off*, the following rule is generated:

$$\begin{array}{ll} \textbf{crl} & \texttt{<}t ::: \mathtt{qid}(s), \mathtt{pre}(\alpha)\texttt{>} \to \texttt{<}t' ::: \mathtt{qid}(s), \mathtt{post}(\alpha)\texttt{>} \\ \textbf{if} & \mathtt{cond}(c_0, \ldots, c_n) \end{array}$$

The function $\mathtt{cond}(c)$ converts the conditions on the transitions to rewriting logic conditions according to the following rules:

$$\mathtt{cond}(c_0, \ldots, c_n) = \mathtt{cond}(c_0) \wedge \cdots \wedge \mathtt{cond}(c_n);$$

For each condition c_i, the conversion is as follows. If c_i is a transition condition (t, α, t'), then it is converted into a conditional rewrite rule. If the step flag is *on*, this rule is as follows:

$$\{t ::: \mathtt{qid}(s), \mathtt{pre}'(\alpha)\} \to [t' ::: \mathtt{qid}(s), \mathtt{post}'(\alpha)]$$

Otherwise the rule is as follows:

$$\texttt{<}t ::: \mathtt{qid}(s), \mathtt{pre}'(\alpha)\texttt{>} \to \texttt{<}t' ::: \mathtt{qid}(s), \mathtt{post}'(\alpha)\texttt{>}$$

where s is the *least sort* applicable to t, therefore implying preregularity.

If c_i is not a conditional transition:

$$\mathtt{cond}(c_i) = c_i$$

The $\mathtt{pre}'$, and $\mathtt{post}'$ functions work in the same way as $\mathtt{pre}$ and $\mathtt{post}$. These equations

follow the equations for the `pre` and `post` functions. The same observations apply.

$$\begin{aligned}
\mathsf{pre}'(\{f_0,\ldots,f_n\}) &= \{\mathsf{pre}'(f_0),\ldots,\mathsf{pre}'(f_n)\}\\
\mathsf{post}'(\{f_0,\ldots,f_n\}) &= \{\mathsf{post}'(f_0),\ldots,\mathsf{post}'(f_n)\}\\
\mathsf{pre}'(U) &= \tilde{U}\\
\mathsf{post}'(U) &= \tilde{U}\\
\mathsf{pre}'(X) &= \tilde{X}\\
\mathsf{post}'(X) &= \tilde{X}'
\end{aligned}$$

For fields the only difference lies on the write-only fields. If i is a read-only index:

$$\begin{aligned}
\mathsf{pre}'(i,c) &= (i,c)\\
\mathsf{post}'(i,c) &= (i,c)
\end{aligned}$$

If i is the unprimed index of a read-write index:

$$\begin{aligned}
\mathsf{pre}'(i,c) &= (i,c)\\
\mathsf{post}'(i,c) &= \epsilon
\end{aligned}$$

If i' is the primed index of a read-write index:

$$\begin{aligned}
\mathsf{pre}'(i',c) &= \epsilon\\
\mathsf{post}'(i',c) &= (i',c)
\end{aligned}$$

If i' is a write-only index:

$$\begin{aligned}
\mathsf{pre}'(i',c) &= (i',())\\
\mathsf{post}'(i',c) &= (i',c)
\end{aligned}$$

Finally, let us describe the practical aspects of this conversion. First, the transitions must be de-sugared and normalized according to this: all relations `-->` are converted to `-{U}->`, the metavariable '`-`' in label expressions is converted to '`U`', and the metavariable '`...`' to '`X`'.

The main function for the conversion from MSDF transitions to rewrite rules is '`make-rewriting-rules`' on module '`MSOS-RULE-GENERATION`'. It has the following signature:

$$\texttt{make-rewriting-rules} : \texttt{Set<Transition>} \times \texttt{LabelType} \times \texttt{CFlags} \rightarrow \texttt{RuleSet}$$

The parameters are: '`Set<Transition>`' is the metarepresentation of a set of MSDF transitions, '`LabelType`', which is the metarepresentation of the label definition present on the same module that contains the set of transitions, and '`CFlags`' is are compilation flags. The image is '`RuleSet`', which is the metarepresentation of a set of rewrite rules. This function iterates over the transitions in '`Set<Transition>`'. In order to generate the beginning and end configurations, it calls the '`lhs-conf`' and '`rhs-conf`' operators,

defined on module ‘AUX-CONF-UTILS’, with the following signature:

$$\texttt{lhs-conf} : \texttt{Term} \times \texttt{QidTerm} \times \texttt{ConfLocation} \times \texttt{CFlags} \to \texttt{Term}$$

‘rhs-conf’ has the same signature. Here, following the order of the parameters, ‘Term’ is the syntactic tree at the beginning of the transition, ‘Qid’ is the type of that syntactic tree, ‘Term’ is the resulting syntactic tree of the transition, ‘ConfLocation’ indicates whether this configuration is being generated in on a conclusion or on a condition, and finally, ‘CFlags’ are the compilation flags. The image sort is ‘Term’, which is a metarepresentation of an MRS condition.

Finally, the conversion of conditions is handled by the function ‘convert-condition’, on module ‘MSOS-RULE-GENERATION’, with the following signature:

$$\texttt{convert-condition} : \texttt{MSOS-Condition} \times \texttt{IsComp?} \times \texttt{CFlags} \to \texttt{Condition}$$

The parameter ‘MSOS-Condition’ is the metarepresentation the conjunction of conditions of a particular transition, ‘IsComp?’ is a flag that indicates if the transition in which these conditions appear uses label composition to that it can be dealt accordingly, ‘CFlags’ is the usual compilation flags and the image sort ‘Condition’ is the metarepresentation of rewriting logic’s conditions.

Source dependent variables in MMT are not checked in MSDF transitions, but *after* the compilation is done—on the generated rewrite rules. This is to simplify the verification process, since the compilation process itself may introduce new variables on the rewrite rules and it is straightforward to check for source dependency on unlabeled transitions, as is the case of rewrite rules (labels in rewrite rules are only decorative and may not contain metavariables).

The verification process simply follows the definition of source-dependent variables: all variables on the source of the conclusion are source-dependent; if all variables on the source of a condition are source-dependent then all the variables on the conclusion are source-dependent. To formalize this, consider a rewriting rule r as:

$$t \to t' \Leftarrow c_1 \wedge \cdots \wedge c_n$$

Let $vars(t)$ be the set of variables in term t. Let $lhs(t)$ be the set of variables in the left-hand side of t and $rhs(t)$ the set of variables on its right hand side. Then, we define incrementally $sd(r)$, the set of source-dependent variables of a rule r, as follows:

- add $vars(t)$ to $sd(r)$;
- for each condition c_i, from left to right, we proceed as follows: if $lhs(c_i) \subseteq sd(r)$ then add $rhs(c_i)$ to $sd(r)$;
- add $lhs(c_i)$ to $sd(r)$ if c_i is a *matching equation* (since the left hand side of c_i in this case is being *instantiated*).

The verification of source-dependent variables is made by function '`source-dependent`', defined in module '`SOURCE-DEPENDENCY-CHECK`', with the following signature:

$$\texttt{source-dependent} : \texttt{Rule} \rightarrow \texttt{Set<Var>}$$

To check whether a rule is source-dependent or not we first select all variables from the rule and then remove those that are source-dependent. There should be no variable left.

Chapter 6

Case studies

This Chapter describes several applications of the MMT: Section 6.1 describes Constructive MSOS [52] and how it is used on the formal definition of programming languages; Section 6.2 describes the semantics of a normal-order language, Mini-Freja [56]; Section 6.3 describes several different distributed algorithms from [36] specified in MSDF and verified in Maude.

6.1 Constructive MSOS

Mosses's Constructive MSOS (CMSOS) [52] is a technique for the specification of programming languages semantics based on the idea that each language construction should be specified in a separated module and the complete language specification is based on the combination of all these modules. A related approach, named Constructive Action Semantics, which uses the same ideas and is developed under the Action Semantics framework, is described in [19, 30, 31].

A further generalization that Constructive MSOS makes is to use *neutral* constructions with regard to a particular programming language. This allows a high degree of reuse of those modules on a wide range of programming languages. Mosses's lecture notes [52] contain a proposed set of these abstract constructions, which we have implemented and call it also as CMSOS. We also exemplify the use of these abstract constructions to give the formal semantics of two different programming languages: a subset ML described in [52] (Section 6.1.2) and a subset of Java, called MiniJava, based on the language described in [2] (Section 6.1.3).

6.1.1 The CMSOS constructions

This Section describes Mosses's proposed set of CMSOS constructions. Each subsection describes a particular aspect of the CMSOS language. We follow [52] on the order of presentation, which divides the semantics of CMSOS into five aspects: expressions, declarations, abstractions, commands, and concurrency. We opted not to present all of CMSOS here, selecting instead those constructions that are needed on the semantics of ML and MiniJava.

A word on terminology of CMSOS modules: recall from Chapter 3 that Mosses's MSOS Tool library separates each module into a file in its own directory. For example, the directory '`Cons/`' contains all the *constructions* of CMSOS. Inside '`Cons/`', the directory '`Exp/`' contains all the constructions related to *expressions*. Inside '`Exp/`', one of the several directories is '`tup/`', which contains the module that defines the syntax of general tuples. MSDF modules must have a name, so we opted to use the directory hierarchy as the name of the module. Also, Mosses divides each construction into *three* modules: one for the abstract syntax, one for the static semantics, and another for the dynamic semantics. To make things simple, we opted to combine both abstract syntax and dynamic semantics in the modules implemented with MMT.

This Section describes a few selected modules from the complete CMSOS specification needed to give the semantics for the other languages in this Chapter, namely ML (Section 6.1.2) and MiniJava (Section 6.1.3). Appendix A contains the remainder of the constructions.

6.1.1.1 Expressions

Expressions are the basic building blocks of CMSOS programs and contain the following constructions: tuples, conditionals, application of operators, and applications of identifiers. We begin by showing the abstract semantics of tuples of expressions first with its most general form of tuples and then a more restricted one.

We begin by defining the general modules that define the set of expressions, '`Exp`', and values, '`Value`'. The module '`Value`' defines the set of values along with a function '`apply-op`', which receives an operation '`Op`', and a sequence of values, '`Value*`'. The set '`Op`', defined on the module '`Cons/Op`', contains all primitive operations on values. To simplify things, we avoid to define rules for extremely obvious operations, such as arithmetic operations, and comparison operations: these are defined outside the specification, as we shall see later. The important thing is that the specifier does not have to worry about defining things like the addition of two integer values, and so on.

```
msos Cons/Op is
 Op .
sosm

msos Value is
 Value .
 Value ::= apply-op (Op, Value*) .
sosm
```

Next, we define the set of expressions, '`Exp`', in the module '`Cons/Exp`'. Expressions evaluate into values, hence the subset inclusion.

```
msos Cons/Exp is
 Exp .
 Exp ::= Value .
sosm
```

The '`Cons/Exp/tup`' below defines a *general* form of tuples in which each element of the tuple is selected nondeterministically to be evaluated.

```
msos Cons/Exp/tup is
 Exp ::= tup Exp* .
 Value ::= tup Value* .

                             Exp -{...}-> Exp'
 -- ---------------------------------------------------------
 (tup (Exp*,Exp,Exp*')) : Exp -{...}-> tup (Exp*,Exp',Exp*') .
sosm
```

A sequential variant has the additional constraint in which elements must be evaluated left to right.

```
msos Cons/Exp/tup-seq is
 Exp ::= tup-seq Exp* .
 Value ::= tup Value* .

                    Exp -{...}-> Exp'
 -- -----------------------------------------------------------
 (tup-seq (Value*, Exp, Exp*)) : Exp
                        -{...}-> (tup-seq (Value*, Exp', Exp*)) .

 (tup-seq Value*) : Exp --> tup (Value*) .
sosm
```

6.1.1.2 Declarations

Declarations constructions declare bindings and evaluate expressions within the context of a set of bindings. A binding associate an identifier to a "bindable" value—a value that may be part of a binding. The set '`Bindable`' is the set of bindable values.

```
msos Data/Bindable is
 Bindable .
sosm
```

Now the set of identifiers '`Id`'. In order to solve a forwarding problem, we first define identifiers in a module '`Id`', as follows:

```
msos Id is
 Id .
 Id ::= Bindable .
sosm
```

We may now define *environments* of bindings, which are used to associated identifiers with bindable values. An environment 'Env' is a parameterized set, a map from 'Id' to 'Bindable'.

```
msos Data/Env is
 Env .

 Env = (Id, Bindable) Map .
sosm
```

The definition of identifiers make use of a read-only component indexed by 'env'. The evaluation of a binding looks up its value on the 'Env' component and returns the value.

```
msos Cons/Id is
 Label = {env : Env, ...} .

     Bindable := lookup (Id, Env)
 -- --------------------------------
 Id : Id -{env = Env, -}-> Bindable .
sosm
```

The set 'Dec' is the set of *declarations* in the language. Declarations evaluate to bindings, hence the subset inclusion.

```
msos Cons/Dec is
 Dec .
 Dec ::= Env .
sosm
```

The construction 'bind' declares the binding of an identifier 'Id' and the value resulted from the evaluation of the expression 'Exp'.

```
msos Cons/Dec/bind is
 Dec ::= bind Id Exp .

                 Exp -{...}-> Exp'
 -- -----------------------------------------
 (bind Id Exp) : Dec -{...}-> (bind Id Exp') .

 (bind Id Value) : Dec --> Id |-> Value .
sosm
```

6.1.1.3 Abstractions

We describe here the constructions associated with procedural abstractions, and recursive bindings. We begin by defining the set of abstractions and the set of its formal parameters.

```
msos Cons/Abs is
 Abs .
sosm

msos Cons/Par is
 Par .
sosm
```

When used to define a language in which abstractions are expressions, the following module must be included.

```
msos Cons/Exp/Abs is
 Exp ::= Abs .
 Value ::= Abs .
sosm
```

The abstraction is defined using the abstract syntax defined in the module '`Cons/Abs/abs-Exp`'. The set '`Passable`' is the set of values that may be used as arguments. The application of an abstraction to a "passable value" creates a '`local`' declaration, with the passable being bound to the parameter with the '`app`' construction described on the module '`Cons/Dec/app`'.

```
msos Cons/Abs/abs-Exp is
 see Cons/Exp/local, Cons/Dec/app .
 see Cons/Exp/Abs, Cons/Exp/app .

 Abs ::= abs Par Exp .

 (app (abs Par Exp) Passable) : Exp -->
                (local (app Par Passable) Exp) .
sosm
```

Closing an abstraction creates a "closure," which is, in essence, an abstraction with an environment.

```
msos Cons/Exp/close is
 see Cons/Abs/closure .

 Exp ::= close Abs .

 Label = {env : Env, ...} .

 (close Abs) : Exp -{env = Env,-}-> (closure Env Abs) .
sosm
```

The module '`Cons/Abs/closure`' contains the definition of the application of a closure to a passable value. Essentially, the environment in the closure is used as an "outer declaration" of the '`local`' construction.

```
msos Cons/Abs/closure is
 see Cons/Exp/local, Cons/Abs .
 see Cons/Exp/app, Cons/Exp/Abs .

 Abs ::= closure Dec Abs .

 (app (closure Dec Abs) Passable) : Exp -->
                   (local Dec (app Abs Passable)) .
sosm
```

6.1.1.4 Commands

Commands are the imperative facet of CMSOS. We begin by defining the set of commands, '`Cmd`'. All commands evaluate to a final value, '`skip`', which is the "do nothing" command and has no meaningful value as an expression.

```
msos Cons/Cmd is
 Cmd .
 Cmd ::= skip .
sosm
```

In order to execute a sequence of commands, we use the '`seq-n`' construction, which takes as parameter the non-empty list of commands, '`Cmd+`', and evaluates each command in order. As each command finishes, that is, evaluates to '`skip`', it is removed from this tuple.

```
msos Cons/Cmd/seq-n is
 Cmd ::= seq Cmd+ .

                  Cmd -{...}-> Cmd'
 -- ------------------------------------------------
 (seq (Cmd, Cmd*)) : Cmd -{...}-> (seq (Cmd', Cmd*)) .

 (seq (skip, Cmd+)) : Cmd --> (seq Cmd+) .

 (seq skip) : Cmd --> skip .
sosm
```

The '`effect`' construction is the bridge between expressions and commands. It evaluates the expression to a value, and disposes of that value. Hopefully, the expression will have changed some read-write or write-only component, like altering a value in storage, or printing a value. Once a final value is produced by the evaluation of the expression, the command evaluates to '`skip`'.

```
msos Cons/Cmd/effect is
 Cmd ::= effect Exp .
```

```
              Exp -{...}-> Exp'
 -- ---------------------------------------
 (effect Exp) : Cmd -{...}-> effect Exp' .

 (effect Value) : Cmd --> skip .
sosm
```

6.1.1.5 Concurrency

Now we define the concurrency aspect of CMSOS specifications. Let us begin by defining the concept of "complete" concurrent programs, or "systems," represented by the set 'Sys'.

```
msos Cons/Sys is
 Sys .
sosm
```

The pool of threads running in a system is bound together by the 'conc' operator. Two rules select nondeterministically which projection should be evaluated at each step. We opted to follow Mosses's specification to the letter here. Had we defined 'conc' a commutative function, only one rule would be necessary. In either case the outcome is the definition of an interleaving model of concurrency.

```
msos Cons/Sys/conc is
 Sys ::= conc Sys Sys .

                   Sys1 -{...}-> Sys1'
 -- ------------------------------------------------
 (conc Sys1 Sys2) : Sys -{...}-> (conc Sys1' Sys2) .

                   Sys2 -{...}-> Sys2'
 -- ------------------------------------------------
 (conc Sys1 Sys2) : Sys -{...}-> (conc Sys1 Sys2') .
sosm
```

6.1.2 ML

ML is a subset of Concurrent ML [61] and is the first example of how to give the semantics of a language in terms of CMSOS. The conversion from ML to CMSOS described in this Section follows the one present in [52].

The transformation from ML's syntax to CMSOS abstract syntax is presented using an equation, where the CMSOS equivalent of an ML construction `c` is denoted between double-bracket parentheses (such as $[\![\mathtt{c}]\!]$). The equation specifies how each component of a construction is translated to CMSOS constructions. The transformation of an ML

construction f that contains parameters x and y is written as: $[\![\mathtt{f(x,y)}]\!] = \mathtt{m}([\![x]\!],[\![y]\!])$ meaning that the f construction is converted into an CMSOS construction 'm' that receives the converted parameters x and y from f.

This Section presents only a subset of the compilation from ML into CMSOS, and Appendix B gives the complete specification.

6.1.2.1 Expressions

We begin by describing ML expressions and their CMSOS counterparts. First we need to gather all CMSOS modules that are necessary for the definitions of expressions in ML. This is done by creating a module 'Lang/ML/Exp' as follows. The module contains explicit references to all CMSOS constructions needed. It also defines that the set of values ('Value') and operators ('Op') are "bindable" in environments, that the set of values are "passable" to procedural abstractions, and that the set of operators contains the constants 'plus', 'times', etc.

```
msos Lang/ML/Exp is
 see Cons/Prog, Cons/Prog/Exp .

 see Cons/Exp, Cons/Exp/Boolean, Cons/Exp/Int,
     Cons/Exp/Id, Cons/Exp/cond, Cons/Exp/app-Op,
     Cons/Exp/app-Id, Cons/Exp/tup, Cons/Exp/tup-seq .

 see Cons/Arg, Cons/Arg/Exp .

 see Cons/Op .

 see Cons/Id .

 Bindable ::= Value | Op .

 Op ::= plus | times | minus | eq | lt | gt .

 Passable ::= Value .
sosm
```

The following module contains the initial dynamic basis for ML expressions. It is defined as a system module that includes the MSDF module 'Lang/ML/Exp'. Recall that we must define the operation 'apply-op' externally and this is done here for each 'Op' constant declared on the 'Lang/ML/Exp' module. Following, we create the initial environment with the default associations of identifiers to operators. We reuse the names of the operator as identifiers, created with the coercion function 'ide' on the mapping.

```
mod Lang/ML/Exp' is
 including Lang/ML/Exp .
 including QID .
```

```
 vars i1 i2 : Int .

 eq apply-op (plus, (i1, i2)) = i1 + i2 .
 eq apply-op (minus, (i1, i2)) = i1 - i2 .
 eq apply-op (times, (i1, i2)) = i1 * i2 .
 eq apply-op (eq, (i1, i2)) = if i1 == i2 then tt else ff fi .
 eq apply-op (lt, (i1, i2)) = if i1 < i2 then tt else ff fi .
 eq apply-op (gt, (i1, i2)) = if i1 > i2 then tt else ff fi .

 op ide : Qid -> Id .
 op ide : Op -> Id .
 op op : Qid -> Op .

 eq init-env = (ide(eq) |-> eq +++ ide(lt) |-> lt +++
                ide(gt) |-> gt +++ ide(plus)  |-> plus +++
                ide(times) |-> times +++ ide(minus) |-> minus) .


 eq op ('+) = plus .   eq op ('*) = times .
 eq op ('-) = minus .  eq op ('<) = lt .
 eq op ('>) = gt .     eq op ('=) = eq .
endm
```

► *Complete expressions*

These are the rules for complete expressions in ML. We have omitted the rules for the conversion of identifiers, special constants (i.e., numbers), application expressions, and infix expressions.

$$
\begin{aligned}
\langle \mathsf{exp} \rangle \rightarrow\ & \langle \mathsf{exp} \rangle \text{ `}\mathtt{andalso}\text{' } \langle \mathsf{exp} \rangle \mid \langle \mathsf{exp} \rangle \text{ `}\mathtt{orelse}\text{' } \langle \mathsf{exp} \rangle \\
& \mid \text{`}\mathtt{if}\text{' } \langle \mathsf{exp} \rangle \text{ `}\mathtt{then}\text{' } \langle \mathsf{exp} \rangle \text{ `}\mathtt{else}\text{' } \langle \mathsf{exp} \rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![e_0 \ \mathtt{andalso}\ e_1]\!] &= \mathtt{cond}\ [\![e_0]\!]\ [\![e_1]\!]\ \mathtt{ff} \\
[\![e_0 \ \mathtt{orelse}\ e_1]\!] &= \mathtt{cond}\ [\![e_0]\!]\ \mathtt{tt}\ [\![e_1]\!] \\
[\![\mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\!] &= \mathtt{cond}\ [\![e_0]\!]\ [\![e_1]\!]\ [\![e_2]\!]
\end{aligned}
$$

6.1.2.2 Declarations

For declarations, let us introduce the relevant MSDF module that contains all CMSOS constructions related to declarations in ML.

```
msos Lang/ML/Dec is
 see Lang/ML/Exp' .
```

```
 see Cons/Prog, Cons/Prog/Dec .

 see Cons/Dec, Cons/Dec/bind, Cons/Dec/simult-seq,
     Cons/Dec/accum, Cons/Dec/local .

 see Cons/Exp, Cons/Exp/local .
sosm
```

▶ *let-Expressions*

We begin by extending the atomic expressions with the '`let-in-end`' expression, which is mapped into the CMSOS '`local`'.

⟨ exp ⟩ → '`let`' ⟨ dec ⟩ '`in`' ⟨ exp ⟩ '`end`'

Let d range over ⟨ dec ⟩.

$[\![\texttt{let d in } e \texttt{ end}]\!] = \texttt{local } [\![\mathrm{d}]\!]\ [\![e]\!]$

▶ *Value bindings*

Next, the declarations are defined. Value bindings are converted into the CMSOS '`bind`' construction.

⟨ dec ⟩ → '`val`' ⟨ vid ⟩ '`=`' ⟨ exp ⟩

$[\![\texttt{val i = } e]\!] = \texttt{bind } [\![\mathrm{i}]\!]\ [\![e]\!]$

6.1.2.3 Imperatives

ML does not have the concept of a "command," as everything is an expression, but it does have imperative features. We have opted to show here the conversion rules for the assignment of values and the looping command.

The module '`Lang/ML/Cmd`' defines the "commands" in the ML language:

```
msos Lang/ML/Cmd is
 see Lang/ML/Dec .

 see Cons/Cmd, Cons/Cmd/seq-n, Cons/Cmd/effect, Cons/Cmd/while .

 see Cons/Exp, Cons/Exp/seq-Cmd-Exp, Cons/Exp/seq-Exp-Cmd,
     Cons/Exp/assign-seq, Cons/Exp/ref, Cons/Exp/assigned .
```

```
 see Cons/Var, Cons/Var/alloc, Cons/Var/deref .

 Storable ::= Value .
sosm
```

Next we add another "external" definition, which is the equation that allocates a new cell on a given store.

```
mod Lang/ML/Cmd' is
 including Lang/ML/Cmd .

 var Store : Store .
 eq new-cell (Store) = cell (length (Store) + 1) .
endm
```

► *Assignment*

Since an assignment in ML does not have any final value, we use the construction '`seq-Cmd-Exp`' to first execute the assignment and then to return the empty tuple ('`tup()`'). The assignment itself is made using the '`assign-seq`' construction by first dereferencing the assigned expression.

$\langle\,\mathsf{exp}\,\rangle \longrightarrow \langle\,\mathsf{exp}\,\rangle$ '`:=`' $\langle\,\mathsf{exp}\,\rangle$

$[\![e_0 \texttt{ := } e_1]\!] = \texttt{seq (effect (assign-seq (deref } [\![e_0]\!]\texttt{) } [\![e_1]\!]\texttt{)) tup()}$

► *Loops*

Finally, we add a construction that is typical of imperative languages, which is the looping command.

$\langle\,\mathsf{exp}\,\rangle \longrightarrow \langle\,\mathsf{while}\,\rangle\ \langle\,\mathsf{exp}\,\rangle$ '`do`' $\langle\,\mathsf{exp}\,\rangle$

$[\![\texttt{while } e_0 \texttt{ do } e_1]\!] = \texttt{seq (while } [\![e_0]\!] \texttt{ (effect } [\![e_1]\!]\texttt{)) tup()}$

6.1.2.4 Abstractions

Let us introduce the relevant MSDF module, '`Lang/ML/Abs`', to introduce the equations for abstractions.

```
msos Lang/ML/Abs is
 see Lang/ML/Dec .

 see Cons/Exp, Cons/Exp/Abs, Cons/Exp/close, Cons/Exp/app-seq .
 see Cons/Abs, Cons/Abs/abs-Exp, Cons/Abs/closure .
 see Cons/Par, Cons/Par/bind, Cons/Par/tup .
 see Cons/Dec, Cons/Dec/app, Cons/Dec/rec .
sosm
```

▶ *Recursive functions*

The version of recursive functions shown here is a very simple form that does not make use of more complicated pattern matching capabilities as we opted to show an example of pattern matching rules in the semantics of the Mini-Freja language, Appendix D. The new option for the ⟨ dec ⟩ nonterminal shows the syntax of recursive functions: the first ⟨ vid ⟩ is the name of the function, the second is the single argument, and the ⟨ exp ⟩ is the body. It is converted into the binding of the function name to a closure.

⟨ dec ⟩ → ‘`fun`’ ⟨ vid ⟩ ⟨ vid ⟩ ‘`=`’ ⟨ exp ⟩

$$[\![\texttt{fun } \mathtt{i}_0 \ \mathtt{i}_1 \texttt{ = } e]\!] = \texttt{rec (bind } [\![\mathtt{i}_0]\!] \texttt{ (close (abs (bind } [\![\mathtt{i}_1]\!]\texttt{) } [\![e]\!]\texttt{)))}$$

6.1.2.5 Concurrency

Finally, let us present the concurrency primitives of ML. This Section shows the primitives for creating new threads and for complete concurrent ML programs.

The module ‘`Lang/ML/Conc`’ gathers the necessary MSDF modules.

```
msos Lang/ML/Conc is
 see Lang/ML/Cmd', Lang/ML/Abs .

 see Cons/Cmd, Cons/Cmd/send-chan-seq,
     Cons/Cmd/start .

 see Cons/Exp, Cons/Exp/recv-chan,
     Cons/Exp/alloc-chan .

 see Cons/Sys, Cons/Sys/Cmd, Cons/Sys/conc,
     Cons/Sys/conc-chan, Cons/Sys/quiet .
sosm
```

▶ *Creating new threads*

The operation ‘`spawn`’ creates a new thread of execution, and is equivalent to the ‘`start`’ construction from CMSOS.

$\langle$ `exp` $\rangle \rightarrow$ '`spawn`' $\langle$ `exp` $\rangle$

$[\![$`spawn` $e]\!]$ = `seq (start` $[\![e]\!]$`)`

► *Complete concurrent ML programs*

Now, the final rule. All ML programs that are concurrent must be prefixed by '`cml`'. It is converted to the '`quiet`' CMSOS construction.

$\langle$ `exp` $\rangle \rightarrow$ '`cml`' $\langle$ `exp` $\rangle$

$[\![$`cml` $e]\!]$ = `quiet (effect` $[\![e]\!]$`)`

Let us discuss the actual implementation of the parser and the conversion from ML to CMSOS. The syntax of ML is given using a Bison grammar and its semantics is presented as a series of MSDF modules. The use of a Bison parser generator is due to some limitations of the Maude interpreter, discussed on Chapter 7.

We use the Bison productions to generate the CMSOS output for each ML construction, using the '`format`' function, which is a function that is similar to the C function '`sprintf`', with the difference that it allocates a pointer and returns the string created according to the formatting specified. For example, let us show the actual rules for the conversion of conditional expressions in ML: The symbol '`$$`' is defined as "the semantic value of the left-hand side of the rule" by the Bison manual. In this specification it will hold the string that contains the translation from ML into CMSOS. The symbol '`$1`' refers to the matched token on the rule.

```
exp: infexp
 | exp ANDALSO exp { $$ = format ("(cond %s %s ff)", $1, $3); }
 | exp ORELSE exp  { $$ = format ("(cond %s tt %s)", $1, $3); }
 | IF exp THEN exp ELSE exp
             { $$ = format ("(cond %s %s %s)", $2, $4, $6); };
```

When the parser reaches the topmost production on the grammar, the output string is the converted CMSOS code. The grammar of this version of ML is simple enough so that there was no need to use an abstract version in the translation to CMSOS, as it was the case with the MiniJava language (Section 6.1.3).

6.1.2.6 Example

As an example of this semantics, let us analyze a concurrent program. It starts as a single thread that creates a channel through the declaration '`chan`' and binds it to the variable '`c`'; then, in turn, spawns three new threads: the first sends the value 10 to the channel

'c', the second sends the value 20 through the same channel, while the third expects to receive a value through 'c'. If we search through all possible outcomes of this program it is expected that there are two final states: one in which the first thread had a successful synchronization with the third thread and another in which the second thread was the successful one.

```
cml
 let chan c in
     (spawn (fn x => send (c, 10)) ;
      spawn (fn x => send (c, 20)) ;
      spawn (fn x => receive c))
 end
```

After the conversion to CMSOS, we have the following program.

```
exec ((quiet
(effect
 (local (bind ide('c) alloc-chan)
  (seq (seq ((effect
   (seq (start (close (abs (bind (ide('x)))
        (seq (send-chan-seq ide('c) 10) (tup ())))))) (tup ()))),
   (effect (seq (start (close (abs (bind (ide('x)))
        (seq (send-chan-seq ide('c) 20) (tup ())))))) (tup ())))))
  (seq (start (close (abs (bind (ide('x)))
       (recv-chan ide('c))))) (tup ())))))))) .)
```

By searching through all possible final states using the 'search' command from Maude we arrive at the expected situation. The first solution shows the remaining, unsynchronized thread stopped at the point where it is trying to send the value 10 through the channel, while the second shows the same with the value 20. (As threads end, they are removed from the configuration.)

```
search in CML-INTERPRETER : exec(...) =>!
    C:Conf .

Solution 1
C:Conf <- <
 quiet
  effect (
   local (ide('c)|-> chan 1)
    local (ide('x)|-> tup())
     seq send-chan-seq chan 1 10 tup())
 ::: 'Sys, {chans ={chan 1},env = void, starting' =(),
           event' = (), store =void}
>
```

```
Solution 2
C:Conf <- <
 quiet
  effect (
   local (ide('c)|-> chan 1)
    local (ide('x)|-> tup())
     seq send-chan-seq chan 1 20 tup())
 ::: 'Sys, {chans ={chan 1},env = void, starting' =(),
           event' = (), store =void}
>

No more solutions.
```

6.1.3 MiniJava

Our implementation of MiniJava into the framework of Constructive MSOS follows the idea that a (simple) object-oriented language is, in its essence, an imperative language in which classes are types, and objects are records [62, 57, 34, 60]. The actual implementation of objects is based on [65].

As an introduction, let us begin with a trivial, abstract, mapping and then expand it with more advanced features, such as recursive reference, and object instantiation. A straightforward view of objects is to consider them as records, whose fields are the methods of the object. We may simplify this even further by using tuples and keeping track of which method corresponds to each projection. As an example, let us consider an object specified in some abstract, Java-like, language:

```
object {
  int i;
  int foo() { return i; }
  int bar() { i := i + 1; return i; }
}
```

It may be represented as a closure, namely, a tuple surrounded by an environment.

```
local (bind x 1) (tup-seq f, b)
```

Here `f` and `b` are closures that represent, respectively, methods '`foo()`' and '`bar()`' from the object above. We omit the abstract representation of those for brevity. In this mapping, all fields of the object are declared as bindings that "surround" the tuple. With this scheme, bindings are not allowed external access directly. It is possible, however, to allow direct access to bindings by creating access functions automatically for each field.

In order to call a method in the object, we obtain the desired projection and evaluate it: if we want to call method '`bar()`', we first obtain the second projection, by applying the operation '`nth(1)`' to the tuple, and then evaluate it, applying the resulting closure to its arguments, the empty tuple in this example.

```
app (app nth(1) o) tup()
```

where `o` is an object.

This is a severely limited form of object-orientation: instance methods are not allowed to be recursive, nor access other methods on the same object. We now add a form of self-reference (e.g., '`this`' in Java) so that a method may call other methods inside the same object. This is achieved by adding a recursive function '`self`' which, when evaluated, returns the object. Being recursive, a method may call '`self`' from within itself, thus having access to the other methods on the object, including itself.

```
local
  (accum (bind x 1)
         (rec bind self
           close abs (bind tup()) (tup-seq f, b)))
  (app self tup())
```

The above code means that a closure is bound recursively to the identifier '`self`' (the closure receives no arguments, specified as '`bind tup()`', in CMSOS); the closure code is, as before, the tuple containing the methods. This closure is defined in accumulation of '`bind x 1`' making those fields available to the methods `f` and `b`.

Being a recursive binding, `f` and `b` may also call '`self`'. Self-reference within a method is achieved by calling '`self`' from within the method and evaluating the desired projection.

The remaining issue is how to instantiate an object based on the type declaration of its class. We follow the approach of object cloning using prototype objects, in the tradition of the Self language [67]. The actual implementation is as follows: each class declaration gives rise to a function declaration that, when evaluated, returns a new object. Thus, a class `C` is converted into the following fragment:

```
bind C (close abs (bind tup()) o)
```

where `o` is a prototype object created from the class declaration. Object instantiation is then converted to: `(app C tup())`, which returns a copy of the object `o`.

With this preliminary exposition, we may fully describe the mapping from MiniJava to CMSOS.

The transformation from MiniJava's abstract syntax to CMSOS constructions is presented using the same notation we used for the ML language on Section 6.1.2. The actual implementation of the converter was made using the SableCC framework due to the same reasons we outlined in the description of the ML language semantics. Again, to simplify the exposition we show only selected components of the language, while keeping the entire specification in Appendix C.

6.1.3.1 Expressions

Expressions consist of mathematical operations, identifiers, method invocations (that always return a value), literals, and objects themselves.

⟨ exp ⟩ → ⟨ math operation ⟩ | ⟨ id ⟩ | ⟨ method invocation ⟩
  | ⟨ literal ⟩ | ⟨ this ⟩ | ⟨ new ⟩

► *Math operations*

⟨ math operation ⟩ → ⟨ exp ⟩ ⟨ math op ⟩ ⟨ exp ⟩
⟨ math op ⟩ → ‘&&’ | ‘<’ | ‘+’ | ‘/’ | ‘%’ | ‘−’ | ‘*’ | ‘>’ | ‘<=’ | ‘>=’ | ‘=’

Let e_i range over ⟨ exp ⟩, and m range over ⟨ math op ⟩.

$$[\![e_0 \;\; m \;\; e_1]\!] \; = \; \texttt{app} \; [\![m]\!] \; \texttt{tup-seq} \; ([\![e_0]\!], \; [\![e_1]\!])$$

► *Method invocations*

⟨ method invocation ⟩ → ⟨ exp ⟩ ‘.’ ⟨ id ⟩ ‘(’ ⟨ exp ⟩*‘)’

$$[\![e \;.\; i \;(\; e* \;)]\!] \; = \; \texttt{app (app nth(}n(i)\texttt{)} \; [\![e]\!]\texttt{) (tup-seq p)}$$

The evaluation of e must return an object; $n(i)$ is the method number in the class of the object returned by e, obtained by looking up the method name i in the metaclass information generated in the static analysis phase of the compilation process; and p is constructed as a sequence of ‘`ref (alloc` $[\![e_i]\!]$`)`’ which allocates a new memory entry for each parameter e_i in $e*$.

► *Self-reference*

⟨ this ⟩ → ‘`this`’

$$[\![\texttt{this}]\!] \; = \; \texttt{app} \; \mathit{self} \; \texttt{tup()}$$

► *Object instantiations.*

⟨ new ⟩ → ‘`new`’ ⟨ id ⟩ ‘`()`’

$$[\![\texttt{new} \; i \; \texttt{()}]\!] \; = \; \texttt{app} \; i \; \texttt{tup()}$$

6.1.3.2 Statements

MiniJava contains the usual statements of imperative programming languages: conditionals, loops, output, assignment, etc. We only show here the conversion of the output command.

⟨ statement ⟩ → ⟨ if ⟩ | ⟨ while ⟩ | ⟨ block ⟩ | ⟨ print ⟩ | ⟨ assign ⟩ | ⟨ empty ⟩

▶ *Output*

⟨ print ⟩ → ‘`System.out.println`’ ‘`(`’ ⟨ exp ⟩ ‘`)`’

```
⟦System.out.println ( e )⟧  = print ⟦e⟧
```

6.1.3.3 Classes

▶ *Class declaration*

As described at the beginning of this Section a class declaration defines a “prototype” object, which is a closure where the fields become bindings and the methods become projections of a tuple.

⟨ class declaration ⟩ → ‘`class`’ ⟨ identifier ⟩ ‘`{`’ (⟨ field declaration ⟩)*
(⟨ method declaration ⟩)*‘`}`’

Let f_i range over ⟨ field declaration ⟩ and m_i over ⟨ method declaration ⟩.

```
⟦class i { f∗  m∗  }⟧  =
   local
    (accum (accum ⟦f∗⟧)
           (rec (bind self
               close (abs (bind dummy) tup-seq (⟦m⟧)))))
    app self tup()
```

▶ *Method declaration*

A method declaration is converted into a closure with parameters being bound simultaneously by the ‘`tup`’ construction. The closure body is a ‘`local`’ definition with the variable declarations as the declaration part the method body as the expression begin evaluated. We use the ‘`seq-Cmd-Exp`’ command so that the last expression evaluated is returned as the method return value.

$\langle$ method declaration $\rangle \longrightarrow \langle$ type $\rangle$ $\langle$ identifier $\rangle$ '(' ($\langle$ parameter $\rangle$)*')'
'{' ($\langle$ var declaration $\rangle$)*($\langle$ statement $\rangle$)*'return' $\langle$ expression $\rangle$ '}'

Let p_i range over $\langle$ parameter $\rangle$.

$[\![$t i (p∗) { v∗ s∗ return e }$]\!]$ =
close (abs tup($[\![$p∗$]\!]$) (local (accum $[\![$v∗$]\!]$) (seq (seq $[\![$s∗$]\!]$) e)))

6.1.3.4 Example

As an example of the translation, let us consider a class that calculates the factorial of a number. This class implements actually two forms of calculating the factorial: the one implemented by the 'RecFat' is recursive, while the one implemented by 'NonRecFac' is direct, using a loop. The code is as follows:

```
class Factorial
{
  public static void main (String[] arg)
  {
    System.out.println (new Fac().Test (6));
  }
}

class Fac
{
  public int Test (int num)
  {
    int r;
    Fac recfac;
    Fac nonrecfac;
    recfac = this;
    nonrecfac = this;
    return recfac.RecFac(num) - nonrecfac.NonRecFac (num);
  }

  public int RecFac(int num)
  {
    int num_aux;
    if (num < 1) num_aux = 1 ;
    else num_aux = num * (this.RecFac(num-1));
    return num_aux;
  }

  public int NonRecFac (int num)
  {
```

```
    int i;
    int fat;
    i = num;
    fat = 1;
    while (i > 0)
      {
        fat = fat * i;
        i = i - 1;
      }
    return fat;
  }
}
```

And the converted code to CMSOS is:

```
local accum bind Fac close (abs bind @ local accum void
rec (bind self close (abs bind @ tup-seq (close (abs
tup bind num local accum bind r ref alloc 0 accum
bind recfac ref alloc 0 accum bind nonrecfac ref alloc
0 void seq seq (effect (assign-seq deref recfac app self
tup ()),effect (assign-seq deref nonrecfac app self tup
()),skip) app minus tup-seq ((app app nth(1) assigned
deref recfac tup-seq ref alloc assigned deref num),app
app nth(2) assigned deref nonrecfac tup-seq ref alloc assigned
deref num)),close (abs tup bind num local accum
bind num_aux ref alloc 0 void seq seq ((cond app lt tup-seq
(assigned deref num,1) effect (assign-seq deref num_aux
1) effect (assign-seq deref num_aux app times tup-seq (assigned
deref num,app app nth(1) app self tup () tup-seq ref
alloc (app minus tup-seq (assigned deref num,1))))),skip)
assigned deref num_aux),close (abs tup bind num local
accum bind i ref alloc 0 accum bind fat ref alloc 0 void
seq seq (effect (assign-seq deref i assigned
deref num),effect (assign-seq deref fat 1),(while app gt
tup-seq (assigned deref i,0) seq (effect (assign-seq
deref fat app times tup-seq (assigned deref fat,assigned
deref i)),effect (assign-seq deref i app minus tup-seq
(assigned deref i,1)))),skip) assigned deref fat))))
app self tup ()) void app app nth(0) local accum void rec
(bind self close (abs bind @ tup-seq close (abs tup
bind @@ local void seq seq (print (app app nth(0)
app Fac tup () tup-seq ref alloc 6),skip) 0))) app self
tup () tup-seq ref alloc 0
```

Executing it under Maude, the following output is produced:

```
rewrite in MINIJAVAPROGRAM : output(...) .
```

```
rewrites: 177289 in 58209ms cpu (58242ms real)
          (3045 rewrites/second)
result Output: output(0)
```

Now, let summarize the actual implementation. The option to use SableCC, which is also a LALR(1) parser generator, was due to SableCC's support for the transformation from concrete syntax to abstract syntax directly in the grammar file using a somewhat "term rewriting" style, which simplifies the construction of the compiler enormously, since MiniJava has a rather complex concrete grammar. This choice also enabled us to reuse the already existing Java 1.1 grammar created by Etienne Gagnon to create the grammar of MiniJava, which is a sublanguage of Java.

The process is as follows. First, the Java 1.1 grammar was modified to exclude the syntax not supported by MiniJava. Next, an abstract syntax of MiniJava, based on the abstract syntax described in Appel's book was created and the concrete grammar was modified to add the rewriting rules that convert from the concrete to the abstract syntax. When this file is processed, SableCC generates tree-walker classes that uses the *visitor* design pattern. This tree-walker is then implemented by the programmer by creating a subclass that adds the appropriate actions that must be performed as the walker visits each node.

The actual compilation process consists of two phases: the static analysis, where all types are checked, and metadata is generated. The second phase makes another visit to the tree, converting each node into its CMSOS equivalent. As in the case of the ML language, instead of constructing the CMSOS code in Java, we could have just exported the abstract syntax tree and let Maude equations do the conversion. In this first implementation, we opted to leave the MiniJava compiler self-contained.

Let us illustrate this description with a simple example, which shows how arithmetic expressions are compiled. We begin with the abstract syntax of arithmetic expressions. In the fragment shown below, the '`expression`' non terminal has a rule named '`math_operation`'. The rule name is important as it will be used to generate the node classes. An arithmetic expression is two expressions with an infixed mathematical operator, represented by the non-terminal '`math_op`'. The prefixes '`[lh]`' and '`[rh]`' have the purpose of identifying each expression on the rule for code-generation purposes.

```
expression =
 {math_operation} [lh]:expression math_op [rh]:expression

math_op = {and} and | {lt} lt {plus} plus
        | {div} div | {minus} minus | {star} star
```

Now, let us show the concrete syntax for these, along with the rewriting rules that generate the abstract syntax. We show only the rule for "additive expressions," since, due to operator precedence issues, the concrete grammar has *nine* different rules. On the grammar below an '`additive_expression`' is converted to the abstract '`expression`'. It has three alternatives: it is either a '`multiplicative_expression`' (not shown here), in this case it is converted to whatever '`multiplicative_expression`' is converted. It can be also the '`plus`' option, in which a new abstract syntax node '`math_operation`'

is created by converting recursively each side of the additive expression described. The same happens for the 'minus' alternative.

```
additive_expression { -> expression }  =
    {multiplicative_expression}
        multiplicative_expression
         { -> multiplicative_expression.expression } |

    {plus}
        additive_expression plus multiplicative_expression
        { -> New expression.math_operation
                  (additive_expression.expression,
                   New math_op.plus (plus),
                   multiplicative_expression.expression) } |

    {minus}
        additive_expression minus multiplicative_expression
        { -> New expression.math_operation
                  (additive_expression.expression,
                   New math_op.minus (minus),
                   multiplicative_expression.expression) };
```

Now we show the fragment from the tree-walker code that passes through the 'math_operation' node. The name of the method is automatically generated by SableCC and it receives as parameter a node that is the representation of the 'math_operation' production. Through getters and setters, each component of the syntax tree is accessed. The conversion is achieved by keeping a dictionary 'cmsos' that maps every node to its CMSOS equivalent. Thus, by looking up the 'lh' and 'rh' expressions (using the functions 'n.getLH()' and 'n.getRh()'), we obtain the representation of each expression. After we have all this information at hand, we put back into the dictionary the mapping from the current node, the mathematical operation, to its CMSOS equivalent.

```
public
void outAMathOperationExpression (AMathOperationExpression n)
{
  String l, r;
  PMathOp o = n.getMathOp ();

  String[] args = { (String) cmsos.get (n.getMathOp ()),
      (String) cmsos.get (n.getLh ()),
      (String) cmsos.get (n.getRh ()) };

  cmsos.put (n, printf ("(app {0} tup-seq ({1}, {2}))", args));
}
```

6.2 Mini-Freja

Mini-Freja [56] is a pure functional programming language with a normal-order semantics.[1] This specification was implemented with several purposes: it does not use Mosses's CMSOS, rather, giving the semantics for the language directly in MSDF; it does not need any external tools, parsing the language directly—with some limitations, as we shall see; it is a big-step semantics; and, finally, it has pattern matching capabilities.

6.2.1 Abstract Syntax

The main construction of Mini-Freja is the *expression*, denoted by the set 'Exp'.

```
Exp .
Exp ::= fn Var => Exp
      | Primu Exp
      | Exp :: Exp [assoc]
      | Exp Primd Exp
      | if Exp then Exp else Exp
      | Exp Exp
      | rec Exp
      | case Exp of Rules
      | let Decls in Exp .

Exp ::= Var | Const .
```

'Exp :: Exp' is the Mini-Freja syntax of lists, 'Exp Exp' is the traditional syntax for application of expressions, 'rec Exp' defines a *recursive expression*, 'Var' is the set of variables on the language (technically speaking, they are not variables, but identifiers, but we follow the nomenclature of Pettersson), 'Const' is the set of constants, 'Primu' are *unary* primitive operators, and 'Primd' are *binary* primitive operations, defined as follows:

```
Primu .
Primu ::= not | neg .

Primd .
Primd ::= lt | le | eq | ne | ge | gt | and
        | or | plus | minus | times | div | mod .

Const .
Const ::= Int | Boolean | nil .
```

Finally, we present the abstract syntax for the declaration of bindings. The syntax is slightly altered from the original, due to preregularity problems. Instead of binding variables to expression with syntax 'Var = Exp' we use 'Var is Exp'.

[1]Pettersson[56] uses "call-by-name"; we follow the terminology of Reynolds [62].

```
Decls .
Dec  .
Dec ::= Var is Exp .
Decls ::= Dec | Decls Decls [assoc] .
```

6.2.2 Semantics

We now describe the dynamic semantics for the Mini-Freja. This specification is based on Pettersson's and Hartel's specifications of the Mini-Freja language, in big-step operational semantics. Mini-Freja, as we mentioned, is a *normal-order* language, that is, an expression may be "suspended" and is only evaluated until it is clear that its value is needed. The only semantic component needed for this specification is the bindings environment.

```
Label = { env : Env, ... } .
```

We now add the set of values to the specification, which consist basically of constants, closures, lists and suspended expressions. Suspended expressions need an environment to be evaluated in the future.

```
Values .

Value ::= susp (Env, Exp)
        | clo (Env, Var, Exp)
        | cons (Value, Value) [assoc]
        | Const .
```

Values are a subset of expressions. We also add a new expression operator '`force` *e*' that forces the evaluation of a suspended expression *e*.

```
Exp ::= Value | force Exp .
```

Let us begin with the evaluation of lists in Mini-Freja, which are into a sequence of recursive applications of the '`cons`' operator.

```
[cons] (Exp1 :: Exp2) : Exp ={env = Env, -}=>
       cons (susp (Env, Exp1), susp (Env, Exp2)) .
```

The arithmetic operators are evaluated in the traditional big-step manner.

```
           (Exp1 ={X1}=> Value1), (Exp2 ={X2}=> Value2),
                          (Value1 Primd Value2 ={X3}=> Value3)
[prim-app] -- ----------------------------------------------
           (Exp1 Primd Exp2) : Exp ={X1 ; X2 ; X3}=> Value3 .
```

```
(Int1 plus Int2) : Exp ==> (Int1 + Int2) .
(Int1 times Int2) : Exp ==> (Int1 * Int2) .
(Int1 mod Int2) : Exp ==> (Int1 rem Int2) .
(Int1 minus Int2) : Exp ==> _-_(Int1, Int2) .

(Int1 eq Int2) : Exp ==> if (Int1 == Int2) then tt else ff fi .
(Int1 ne Int2) : Exp ==> if (Int1 == Int2) then ff else tt fi .
```

The following rule establishes that canonical forms always evaluate to themselves.

```
Const : Exp ==> Const .
```

In order to evaluate the conditional construction we define an auxiliary operation '`if-choose(b,`e_1`,`e_2`)`', which work as follows: if b is true, it evaluates to e_1, otherwise it evaluates to e_2.

```
Exp ::= if-choose (Value, Exp2, Exp3) .

[if-choose-tt] if-choose (tt, Exp2, Exp3) : Exp ==> Exp2 .
[if-choose-ff] if-choose (ff, Exp2, Exp3) : Exp ==> Exp3 .

     (Exp1 ==> Value),
     (if-choose (Value, Exp2, Exp3) ==> Exp),
     (Exp ==> Value')
[if] -- ---------------------------------------------
     if Exp1 then Exp2 else Exp3 : Exp ==> Value' .
```

Closures evaluate to its value form, '`clo(`ρ`,`v`,`e`)`', consisting on the "captured" environment ρ, the argument v, and the closure expression e.

```
[clo] (fn Var => Exp) : Exp ={env = Env, -}=>
      clo (Env, Var, Exp) .
```

The following rules specify the meaning of the '`force`' operator. Essentially, non-suspended expressions evaluate to themselves. Suspended expressions (rule '`[force-susp]`') '`susp(`ρ`,`e`)`' are evaluated by replacing the current environment with ρ and evaluating e into an intermediate value v, which is itself "forced" into the final value v'.

```
[force-const] force Const : Exp ==> Const .

[force-clo] force clo (Env, Var, Exp) : Exp
            ==> clo (Env, Var, Exp) .

[force-cons] force cons (Value1, Value2) : Exp
        ==> cons (Value1, Value2) .
```

```
        Exp ={env = Env', -}=> Value,
                     force Value ={env = Env, -}=> Value'
[force-susp] -- -----------------------------------------------
        force susp (Env', Exp) : Exp ={env = Env, -}=> Value' .
```

Application of expressions implements a type of β-reduction. It is expected that 'Exp2' on rule '[app]' below evaluates to a closure.

```
      Exp1 ={env = Env, -}=> clo (Env1, Var1, Exp'),
       Env2 := (Var1 |-> susp (Env, Exp2)) / Env1,
        Exp' ={env = Env2, -}=> Value
[app] -- -------------------------------------------
      (Exp1 Exp2) : Exp ={env = Env, -}=> Value .
```

The rule for variables follows the traditional rules, with the additional requirement that the returned value must be "forced."

```
         Value := lookup (Var, Env),
         (force Value ={env = Env, ...}=> Value')
[lookup] -- -----------------------------------
         Var : Exp ={env = Env, ...}=> Value' .
```

The rule for the 'let' operator evaluates the declarations 'Decls' into a set of bindings 'dec(Env')' and evaluates 'Exp', overriding its environment with these bindings.

```
Decls ::= dec (Env) .

      Decls ={env = Env, -}=> dec (Env'),
      Env'' := Env' / Env, Exp ={env = Env'', -}=> Value
[let] -- ----------------------------------------------
      (let Decls in Exp) : Exp ={env = Env, -}=> Value .
```

In order to evaluate 'Decls', each 'Dec' is evaluated in turn the resulting environments are concatenated.

```
         Dec ={env = Env, -}=> dec (Env'),
         Env'' := Env' / Env,
             Decls ={env = Env'', -}=> dec(Env''')
[decls]  -- ---------------------------------------
         (Dec Decls) : Decls ={env = Env, -}=>
                           dec (Env' +++ Env''') .

             Dec ==> dec (Env')
[decls] -- -------------------------
        Dec : Decls ==> dec (Env') .
```

The following rule is necessary to give the normal order evaluation: expressions are not evaluated as they are bound to variables; they are first converted into "suspended" values.

```
[dec] (Var is Exp) : Dec ={env = Env, -}=>
       dec (Var |-> susp (Env, Exp)) .
```

Recursive functions in the language are implemented using a fixed point operator, following ideas present in Reynolds's book [62]. The expression '`Exp`' is expected to be a lambda-expression, such as '`fn v => e`'.

```
              Exp (rec Exp) ==> Value
[fixed-point] -- ----------------------
              rec Exp : Exp ==> Value .
```

The following rule evaluates an expression using the '`exec`' and '`strict`' constructions, the later is similar to the '`force`' construction with the exception that it operates over *values*, while '`force`' operators over *expressions*.

```
Value ::= exec Exp | done Value .

       Exp ==> Value,
                strict Value ==> Value'
[exec] -- ----------------------------
       exec Exp : Exp ==> done Value' .

Value ::= strict Value .

[strict-const] strict Const : Value
               ==> Const .

[strict-clo] strict clo (Env, Var, Exp) : Value
             ==> clo (Env, Var, Exp) .

              strict Value1 ==> Value1',
                             strict Value2 ==> Value2'
[strict-cons] -- ------------------------------------
              strict cons (Value1, Value2) : Value
                         ==> cons (Value1', Value2') .

              Exp ={env = Env', -}=> Value,
              strict Value ={env = Env, -}=> Value'
[strict-susp] -- ---------------------------------
              strict susp (Env', Exp) : Value
                         ={env = Env, -}=> Value' .
```

6.2.3 Example: sieve of Eratosthenes

Let us demonstrate the normal-order characteristics of Mini-Freja by creating a sieve of Eratosthenes using "lazy lists." The algorithm works as follows: we create an infinite list of numbers (function '`from`' below). This infinite list of numbers is filtered to keep only the prime numbers (functions '`filter`', '`sieve`', '`not-div`'). From this infinite list of primes, we take the first few (function '`take`' below). The implementation is as follows. We opted to split each function declaration into its own constant to simplify the exposition. We begin by creating all the constants that are used on the specification. We could have created these constants on an MSOS module but since they are used only for expository purposes and need equations anyway, we opted to declare them in a single Maude module.

```
ops fat n n0 xs0 x y xs pp N filter
    not-div sieve take from primes : -> Var .

op filterd : -> Dec .
op fromd : -> Dec .
op taked : -> Dec .
op not-divd : -> Dec .
op sieved : -> Dec .
op primesd : -> Dec .
op fatd : -> Dec .
```

Function '`filter`' receives as arguments (in curried form) a predicate and a list and returns only the elements from the list that satisfy the given predicate. Due to the *fixed point operator* defined formally in Appendix D, in order to declare a recursive function f, whose contents is an expression e, we write it as '`rec f is fn f =>` e'.

```
eq filterd
= filter is rec (fn filter => fn pp => fn xs0 =>
   case xs0 of
     p nil       => nil
  || p x :: p xs => if (pp x) then
                      x :: ((filter pp) xs)
                    else
                      (filter pp) xs) .
```

Function '`from`' initiates an infinite list beginning at the value specified by its first argument

```
eq fromd
= from is rec (fn from =>
               (fn n => (n :: (from (n plus 1))))) .
```

Function '`take`' receives as arguments a number n and an infinite list l and returns the first n elements from l.

```
eq taked
= take is rec (fn take => fn n0 => fn xs0 =>
   case n0 of (p 0 => nil)
            || p n => (case xs0 of
                         p x :: p xs =>
                          (x :: ((take (n minus 1)) xs))
                       || p nil       => nil)) .
```

The function '`not-div`' is used as a predicate on function '`filter`'. It takes two arguments x and y and returns true if y divides x.

```
eq not-divd = not-div is (fn x => (fn y => ((y mod x) ne 0))) .
```

Function '`sieve`' implements the sieve by removing from the list it receives all numbers that are divisible by the rest of the numbers present on the list. The list must begin with two, for obvious reasons.

```
eq sieved
= sieve is rec (fn sieve => fn xs0 =>
                case xs0 of
                 ((p x :: p xs) =>
                  (x :: (sieve ((filter (not-div x)) xs))))) .

eq primesd = primes is sieve (from 2) .
```

Now, we may execute the program by concatenating all the declarations above and asking for the first 18 primes.

```
rewrite in PRIMES :
  < exec(let filterd taked fromd not-divd sieved primesd N is 18
         in ((take N) primes))::: 'Exp,init-rec >
result Conf :
  < done cons(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,
              59,61,nil)::: 'Exp,{env = void} >
```

6.3 Distributed algorithms

This Section shows the use of the *Maude MSOS Tool* in the specification and verification of distributed algorithms. As we mentioned before, SOS and MSOS are formalisms not only used in the specification of programming languages, but also of concurrent systems [50, 51]. The conversion from MSOS to Rewriting Logic performed by *Maude MSOS Tool* using the Maude interpreter enables the use of Maude's built in Linear Temporal Logic (LTL) model checker and breadth-first search capabilities, detailed in Section 2.3.1.

This Section is organized as follows: Section 6.3.1 defines a model for process execution of distributed processes and Section 6.3.2 shows the examples from [36] and [20]. Appendix E contains the complete specification and several variations of the Dining Philosophers specifications (E.2), and three additional examples of distributed algorithms: a simple mutual exclusion using semaphores (E.1), Lamport's Bakery Algorithm (E.3) and Leader Election (E.4).

6.3.1 Process execution model

This Section outlines a simple process execution model. We begin with the notion of *processes* and *process identifiers*. The set 'Proc' represents processes in our specifications:

```
Proc .
```

A process contains an integer as its process identifier (pid) and an abstract data type that represents its local state ('St'). The local state is dependent on the algorithm being specified, and will be mostly used on our specifications to record the state of the computation of a process, but it can also store temporary values that are local to a specific process throughout the execution.

```
St .
Proc ::= prc (Int, St) .
```

We follow ideas present in [39, 4] and create a set 'Soup' that represents an associative-commutative "soup of processes." A single process is a trivial soup. The evolution of the soup is done by selecting non-deterministically a process out of the "floating processes," made using matching modulo associativity and commutativity, evaluating this process, and putting it back into the soup.

```
Soup ::= Proc .
Soup ::= Soup Soup [assoc comm] .
```

The following rule implements the evolution of the soup of processes.

```
                       Proc -{...}-> Proc'
[exec1] -- ------------------------------------
         (Proc Soup) : Soup -{...}-> Proc' Soup .
```

One could write the left-hand side of the conclusion as 'Soup1 Soup2', instead of 'Proc Soup'. This would select non-deterministically an entire portion of the soup to evolve. This extra generality is not necessary on some of the algorithms shown here, since they specify transitions for a particular process, and not a subset of processes. The alternative rule would then recursively apply to itself until 'Soup1' is a single 'Proc' to which there are other applicable transitions available, generating unnecessary rewrites, and artificially augmenting the state space of a particular specification.

Finally, we need a rule for the trivial case in which the soup consists of a single process:

```
         Proc -{...}-> Proc'
 -- --------------------------
  Proc : Soup -{...}-> Proc' .
```

6.3.1.1 Process communication models

This Section describes two possible models for process communication: shared memory and message-passing.

Shared memory model is trivially implemented with the use of a read-write component on the label to store the shared variables of the processes. The remainder of this Section deals with a simple message passing model on an asynchronous network.

The set '`Msg`' represents the *messages* that circulate on the network.

```
Msg .
```

The specific type of message is, as usual, algorithm dependent, but, for this exposition let us assume the following:

```
Msg ::= msg Int from Int to Int .
```

where the first argument is the *value to be transfered*, the second is the *origin* of the message, and the third is the *destination*.

The message passing mechanism in our specification follows Maude's pattern matching capabilities. In this mechanism, messages and processes "float" on the soup and the transition rules will emulate the transmission of a message to a process by matching the destiny argument of the message with the pid present on the process object. For this we need to expand the range of the '`Soup`' object to allow also messages.

```
Soup ::= Msg | Proc .
```

To exemplify the message passing through matching, consider the following fragment in which a message originating from process '`Int'`' with a destination of process '`Int`' is paired with the process of pid '`Int`'.

```
prc (Int, St) (msg C from Int' to Int)
```

Since now processes *and* messages need to interact for the evolution of the soup, we must generalize the interleaving rule to allow the evolution of a portion of the soup.

```
                Soup1 -{...}-> Soup'1
 [exec2] -- ---------------------------------------
         (Soup1 Soup2) : Soup -{...}-> Soup'1 Soup2 .
```

While it is true that this rule has the drawback discussed at the beginning of Section 6.3.1, its generality allows no *a priori* commitments on the nature of the algorithm. In order words, the relationship of objects and messages is left open for a wide variety of interactions, depending on the specific needs of a particular specification.

6.3.1.2 Justice

Let us discuss justice in rule '`[exec1]`'. It is easy to notice that there is no specific order in which processes are selected to be evaluated: all possible traces of execution are produced, including those in which a particular process loops forever, not letting any other process evolve.

A very simple way of adding justice to a specification is by controlling which process is chosen to be evaluated through some sort of *scheduling policy*. Let us describe one such policy, the round-robin, or *fair* scheduling of processes. It consists of having a counter that operates modulo the number of processes: the current value of the counter is the pid of the process that is allowed to execute; upon executing one step, the counter is incremented. With this strategy, all processes eventually reach their execution turn.

This is implemented by adding a read-write component indexed by '`fair`' to the label.

```
Label = { fair : Int, fair' : Int, ... } .
```

We now change rule '`[exec1]`' to reflect the scheduling just described. Let us assume that there is a constant '`n`' that will be instantiated later, through an equation, with the number of processes in the soup.

```
Int' := (Int + 1) rem n,
prc (Int, St) -{fair = Int, fair' = Int, ...}-> prc (Int, St')
-- --------------------------------------------------------------
(prc (Int, St) Soup) : Soup -{fair = Int, fair' = Int', ...}->
                                              prc (Int, St') Soup .
```

Even though this solution works, it is far too restricted: all processes receive the same probability of execution, which is not always the case, and the processes always execute on the same order. This last restriction may be lifted by using a pseudo-random number generator and randomly selecting which process to evaluate at a time.

6.3.2 Examples

6.3.2.1 Another thread game

Let us begin with a simple specification, based on the *thread game* described in [20]. This specification also demonstrates the problems associated with the justice (or lack thereof) in the specification.

Two threads continuously attempt to update the value of a shared variable: one process increments the value by one, while the other decrements the value by one. This shared variable is modeled using a read-write component indexed by '`v`'.

```
Label = {v : Int, v' : Int, ...} .
```

Let us formalize the behavior of both threads. Process 'prc 0' increments and process 'prc 1' decrements. Let us also limit the value of the shared variable to no less than zero and no more than five, an arbitrary value.

```
   Int < 5, Int' := Int + 1
-- ------------------------------------------------
(prc 0) : Proc -{v = Int, sh' = Int', -}-> prc 0 .

   Int > 0, Int' := Int - 1
-- ------------------------------------------------
(prc 1) : Proc -{v = Int, sh' = Int', -}-> prc 1 .
```

These next two rules keep the system running when the variable is in the established limits.

```
   Int >= 5, Int' := Int
-- ------------------------------------------------
(prc 0) : Proc -{v = Int, sh' = Int', -}-> prc 0 .

   Int <= 0, Int' := Int
-- ------------------------------------------------
(prc 1) : Proc -{v = Int, sh' = Int', -}-> prc 1 .
```

In order to analyze this specification with Maude's model checker (Section 2.3.1.3), let us create a proposition 'max(i)', which holds whenever the shared variable has a value equal or inferior to i.

```
op max : Int -> Prop .

ceq (< S, { v = I', PR } >) |= max (I) = true
if I' <= I .
```

If we use the *fair scheduling* of processes described on Section 6.3.1.2, we will notice that the value of the shared variable will never exceed one.

```
rewrites: 2511 in 26ms cpu (26ms real) (93013 rewrites/second)
reduce in CHECK :
  modelCheck(init,[] max(1))
result Bool :
  true
```

Using the specification without fairness we quickly arrive at a counterexample where process zero always increments the shared variable up to five.

```
reduce in CHECK :
  modelCheck (init, [] max(1))
result ModelCheckResult :
  counterexample(
  { < (prc 0 prc 1), {fair = 0, v = 0} > }
  { < (prc 0 prc 1), {fair = 0, v = 1} > }
  { < (prc 0 prc 1), {fair = 0, v = 2} > }
  { < (prc 0 prc 1), {fair = 0, v = 3} > }
  { < (prc 0 prc 1), {fair = 0, v = 4} > },
  { < (prc 0 prc 1), {fair = 0, v = 5} > })
```

6.3.2.2 Dining Philosophers

This Section presents a solution to Dijkstra's "Dining Philosophers" problem as described in [36]. This solution is based on breaking the symmetry on the moment in which each philosopher acquires its fork: philosophers with even pids first attempt to acquire the fork at their left, while philosophers with odd pids first attempt to acquire the fork at their right.

By definition, the right fork of a philosopher $\mathtt{i}$ has number $\mathtt{i}$, and the left fork has number $\mathtt{i} + 1 \bmod \mathtt{n}$. When there is a competition to acquire a fork, the pids of the competing philosophers are inserted on a queue present in each fork. As each philosopher is done with the fork, it removes its pid from the queue.

The MSDF specification is as follows. First we need to map each fork id to a list of pids to implement the queue on each fork. The set 'Pids' defines that list of pids, while 'Queue' defines the map from integers (fork ids) to 'Pids'. Although a specific queue needs to be shared only between two philosophers, to simplify the specification we opted to make it globally shared by creating a read-write component indexed by 'q'.

```
Pids = (Int) List .
Queue = (Int, Pids) Map .
Label = {q : Queue, q' : Queue, ...} .
```

The specification is parameterized by a constant 'n', which should be instantiated through an equation to the correct number of philosophers on the table.

```
Int ::= n .
```

Each philosopher is a process with the following states. Each state will be detailed on the subsequent transitions.

```
St .
St ::= srem
     | stest-right
     | stest-left
     | sleave-try
```

```
| scrit
| sreset-right
| sreset-left
| sleave-exit
| stry
| sexit .
```

Let us show only the transitions for odd-numbered processes. The even-numbered transitions are symmetric to the ones shown here. Initially, all philosophers are hungry and will attempt to acquire their forks, that is all processes are in the state '`stry`'. Odd-numbered processes, selected with the predicate '`odd(i)`', attempt to acquire their right forks (state '`stest-right`').

```
                      odd (Int)
-- ------------------------------------------------
prc (Int, stry) : Proc --> prc (Int, stest-right) .
```

At this point we make a slight modification to the original algorithm. The original rule is the following: if the fork is unavailable, the process put its pid on the queue, and go back to test if its pid reached the beginning of the queue, as the rule below shows. Recall from Section 4.3 that '`insert-back`' and '`first`' are functions operating on parameterized lists.

```
odd (Int),
Pids := lookup (Int, Queue),
Pids' := if (not Int in Pids)
         then insert-back (Int, Pids) else Pids fi,
Queue' := (Int |-> Pids') / Queue,
St := if first (Pids') == Int
         then stest-left else stest-right fi
-- ----------------------------------------------
prc (Int, stest-right) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, St) .
```

This *busy waiting* makes verification more complex since, if the algorithm is incorrect, it would enter a *livelock* and not in a *deadlock*. Deadlocks are easier to check with Maude: it needs only to look for a state to which no rule applies, since the system is reactive. We opted to change the algorithm by allowing at most one process in the queue, making it behave as a semaphore. The transition will only happen when a process successfully acquires a fork by putting its pid on the queue and immediately checking that it is at the beginning of the queue—that is, the queue was empty. If the fork is successfully acquired, the process moves to acquire its left fork by changing its state to '`stest-left`', otherwise it does not change its state.

```
odd (Int),
Pids := lookup (Int, Queue),
```

```
Pids' := if (not Int in Pids)
         then insert-back (Int, Pids) else Pids fi,
Queue' := (Int |-> Pids') / Queue,   first (Pids') == Int
-- -------------------------------------------------------
prc (Int, stest-right) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, stest-left) .
```

Not only is this rule simpler than the previous one, it also has the advantage of creating a *deadlock* instead of a *livelock* if the specification has any problems.

The remainder of the specification appears on Appendix E.2 and it works as follows: after acquiring its right fork, the process attempts to acquire its left fork using a similar transition. Upon acquiring both forks, a process moves to its critical region and proceeds to put the forks down, first the right, then the left and finally enters its '`srem`' state, which represents a philosopher thinking. The process moves from '`srem`' directly to '`stry`', indicating that right after thinking it becomes hungry again.

Searching for a final state on with the '`search`' command with the '`=>!`' predicate relation is a good way of finding a deadlock on the algorithm, since a final state is a state in which no rule applies, meaning that the entire pool of processes is stopped and cannot continue to evolve.

The auxiliar function '`initial-conf`' creates an initial configuration with the desired number n of philosophers. For $n = 4$, the algorithm takes 3.6 seconds to find that there is no final state, as we expect on a correct configuration.

```
rewrites: 760825 in 3604ms cpu (3646ms real)
          (211079 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

No solution.
```

When $n = 6$, the search takes two minutes.

```
rewrites: 26197002 in 127450ms cpu (127420ms real)
          (205547 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

No solution.
```

We may further test the specification using more searches. For example we know that, in a configuration with four philosophers, two philosophers may eat at the same time (that is, be at their respective '`scrit`' states), however, a philosopher may never eat concurrently with its neighbor. We may verify this by asking '`search`' to return all states in which two philosophers are in their '`scrit`' states:

```
search in SEARCH : initial-conf =>*
 < (prc(I1:Int,scrit)prc(I2:Int,scrit) S:Soup)::: 'Soup,
```

```
   R:Record > .

I1:Int <- 0 ; I2:Int <- 2
I1:Int <- 1 ; I2:Int <- 3
I1:Int <- 2 ; I2:Int <- 0
I1:Int <- 3 ; I2:Int <- 1
```

Another search confirms that three philosophers never eat at the same time in a four-philosopher configuration.

```
search in SEARCH : initial-conf =>*
 <(prc(I1:Int,scrit)prc(I2:Int,scrit)prc(I3:Int,scrit)
   S:Soup)::: 'Soup,R:Record > .

No solution.
```

Because the specification does not have justice, it is possible that a particular philosopher may never have the chance to eat, as the following model checking shows. The proposition '`state(i,s)`' holds when process `i` is in state `s`. Looking at the counterexample, we notice that process '`0`' is "stuck" in '`sleave-try`' while process '`2`' keeps entering and leaving its critical region indefinitely.

```
rewrites: 3539 in 60ms cpu (60ms real) (58983 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state(0,scrit))
result ModelCheckResult :
  counterexample(
  { prc(0,stry) prc(1,stry) prc(2,stry) prc(3,stry)}...,

  { prc(0,sleave-try) prc(2,srem) ... }
  { prc(0,sleave-try) prc(2,stry) ... }
  { prc(0,sleave-try) prc(2,stest-left) ... }
  { prc(0,sleave-try) prc(2,stest-right) ... }
  { prc(0,sleave-try) prc(2,sleave-try) ... }
  { prc(0,sleave-try) prc(2,scrit) ... }
  { prc(0,sleave-try) prc(2,sexit) ... }
  { prc(0,sleave-try) prc(2,sreset-left) ... }
  { prc(0,sleave-try) prc(2,sreset-right) ... }
  { prc(0,sleave-try) prc(2,sleave-exit) ... }
```

Appendix E.2 completes this Section with further variations on the specification for the solution to the Dining Philosophers, such as a specification in which, after eating each philosopher stops, and a specification with a fair scheduling of the processes. We also specify an *incorrect* version of the algorithm and use Maude's formal tools to detect the deadlock.

Chapter 7

Conclusion

This Chapter discusses our work showing our contributions, and outlines some limitations of the current implementation of *Maude MSOS Tool* and outlines future work. It is organized as follows. Section 7.1 discusses some limitations of *Maude MSOS Tool*; Section 7.2 discusses possible enhancements to the tool; while Section 7.3 outlines the contributions of our work.

7.1 Design decisions and limitations

Section 7.1.1 explains our choice of using a default *least set* for typed syntactic trees in the conditions of MSDF transitions; Section 7.1.2 describes some problems in MSDF parsing that made it somewhat different from Mosses's proposed MSDF; Section 7.1.3 explains an undesirable side-effect of our choice of compilation process; Section 7.1.4 describes some design choices of Maude that affect negatively MMT; finally, Section 7.1.5 describes why alternatives to our choice of automatic metavariable typing cannot be implemented in an order-sorted equational theory.

7.1.1 Typed syntactic trees in conditions

The use of a default least sort as the type of the syntax tree in the conditions of MSDF rules (Section 4.2.4) could be generalized to the same syntax of typed syntax trees in conclusions. However, pragmatically it seems that the current approach is sufficient, given the several examples that were built using it. It may be because, in structural operational semantics, often, if not always, the condition involves a very specific subtree of the general tree at the conclusion, which has the least possible set.

7.1.2 Limitations of the MSDF syntax in MMT

The Maude 2.1.1 parser allows the specification of highly flexible syntax, but few punctuation symbols are used to separate identifiers, forcing the user to insert arbitrary spaces to avoid parsing errors. For example, it is common in languages to declare an assignment operator such as '`_:=_`', but it is not possible to write '`x:=y`', as it would be identified as

an erroneous "constant." The same problem does not happen when symbols '`{`', '`}`', '`,`', '`(`', '`)`', among others, are used: it is possible to write '`{({z,(x,y)})}`' without having to insert arbitrary spaces.

Another problem is that, currently, there is no way of specifying the "form" of some tokens (e.g., that its first character should be an uppercase vowel). A general solution would be to add some form of lexical parser generator in Maude that combines the flexibility of regular expressions with the intended algebraic semantics of Maude operators.

With some programming effort these restrictions may be removed due to the reflective nature of Maude. In order to deal with some limitations of spacing requirements, one could easily add a "filter," with signature `QidList` $\rightarrow$ `QidList`, on the input stream coming from '`LOOP-MODE`' that "breaks" input tokens in certain specific characters. For example, an input such as '`x:=y`' would be transformed by '`LOOP-MODE`' into '`'x:=y`' and this function would break it into '`'x ':= 'y`', according to some rule that makes any sequence of punctuation symbols a token separator. A meta-level solution is also applicable to overcome the limitations due to the lack of regular expressions: after inputting the user text and parsing into a data type, one could write a static checker at the meta-level that rejects ill-formed programs, by effectively *programming* what a regular expression would match.

Moreover, when designing the concrete grammar of a programming language that imports built-in modules, often conflicts arise between predefined operators and those present on the language, as the following example show:

```
fmod PROBLEM1 is
 inc INT .

 sorts Value Exp Id .
 subsort Value < Exp .
 subsort Id < Exp .
 subsort Int < Value .

 op _+_ : Exp Exp -> Exp [ditto] .
 op _-_ : Exp Exp -> Exp [ditto] .
 op _<_ : Exp Exp -> Exp .
endfm
```

Loading this module into Maude, we receive the following warning:

```
Warning: "limitations.maude", line 23 (fmod PROBLEM1):
    declaration for _<_ has the same domain kinds as the
    declaration on "prelude.maude", line 190 (fmod NAT)
    but a different range kind.
```

because, in module '`NAT`' (built-in), the operator '`_<_`' has image sort '`Bool`', disconnected from the image sort '`Exp`' of '`_<_`'. Also, in order to suppress another warning we had to use the attribute '`ditto`' on the operations '`_+_`', and '`_-_`', which means that the attributes

of these operations are the same as the operations defined for sorts that are related to 'Exp', such as 'Nat', 'Int', etc., that is, '[assoc comm prec 33]'. Apparently, there is no good solution to this, apart from modifying the built-in operations to use less "natural" names such as 'ADD' instead of '_+_'. Of course, this solution would break compatibility of all specifications written so far and is not really recommended. Normally, to solve these problems, one isolates built-in sorts such as 'Int' by using coercion functions to artificial sorts (say, 'Integers'). The simplest solution is not to use the conflicting symbols.

All these restrictions cater for a difference in Mosses's MSOS Tool and *Maude MSOS Tool* and the use of Bison and SableCC to define formally the syntax of the programming languages ML and MiniJava. Some of the differences from Mosses's tool to ours are: (i) the line of dashes that separates the conclusion from the premises must necessarily have '--', followed by a space before the rest of the line is written. In Maude, three dashes ('---') start a line comment (ii) restrictions on spacing while writing label expressions and value added syntax trees. Other restrictions, described in Section 7.1.4, include the use of named modules and preregularity problems arising from the compilation process.

7.1.3 Loading of modules

Maude MSOS Tool follows Full Maude in the way it handles module hierarchies: modules included are incorporated into the final, compiled module, called a "flat module." This has the drawback of slowing down the loading of modules, as in the case of the CMSOS specification (of the order of 100 modules), since, as modules keep including other modules, their flattened versions (metarepresented in Full Maude's database) keep growing. This also happens in Full Maude for the same reason, and a simple experiment confirms this: generating 100 modules at random, with random module inclusions, but with the exact (meaningless) contents, such as this (the numbers vary from module to module):

```
(omod name98 is
  including name47 .
  including name10 .
  including name79 .

  sort s98 .

  ops c98 d98 : -> s98 .
  ops x98 y98 : s98 s98 -> s98 .

  eq x98(c98,d98) = d98 .

  vars W WW : s98 .

  crl y98(W,WW) => d98 if W => WW .
endom)
```

we obtain the following figures: the first module loaded takes around 2000 rewrites and 240 milliseconds to be processed by Full Maude. The final modules, depending on the number

of inclusions that they carry (no more than ten are generated), take up to *26 million rewrites* and *90 seconds* to load, with the same content as the one shown above. We used "object-oriented" ('`omod ... endom`') modules in Full Maude since, as MMT, they are compiled into system modules, although the compilation is somewhat straightforward since it does not have to deal with a different language for the syntax definition.

The example shows an extremely connected modular specification, but a moderately connected specification such as CMSOS (Section 6.1, Appendix A) in MMT may cause problems as well. For example, the module that loads all the necessary CMSOS modules to give the semantics of the ML language (Section 6.1.2) takes around 950000 rewrites and 16 seconds to process.

The same test made using only the Maude interpreter shows that it does not have this limitation. Unfortunately, Maude 2.1.1 does not have an operation that loads a metarepresented module and, if *Maude MSOS Tool* is to be modified to run under Maude directly (instead of Full Maude) for any reason, it would not have the ability of, in a *single session* load and execute MSDF specifications—it would have to work as a "preprocessor" that, on the first execution, outputs Maude code from MSDF specifications into a file and, on a second execution, loads this file to execute those specifications. All this complexity could be encapsulated by a shell script that hides this preprocessing step, even though at this point the tool ceases to be a formal tool at least in this particular aspect.

7.1.4 Limitations on the generality of MSDF in MMT

Membership equational logic has the prerequisite that a module should be preregular [41], that is, each term `t` should have a *least sort*. This algebraic requirement may affect modularity, as the following, taken from [28], shows. Modules '`SIG0`' and '`SIG2`' are, by themselves, preregular.

```
fmod SIG0 is
 sorts t1 t2 .
endfm

fmod SIG2 is
 including SIG0 .
 op f : t1 -> t1 .
 op f : t2 -> t2 .
endfm
```

Combining both modules with '`SIG`' below makes '`f`' non-preregular, with an additional advisory from the Maude interpreter.

```
fmod SIG is
 including SIG2 .
 sort s .
 subsort s < t1 .
 subsort s < t2 .
```

```
endfm

Advisory: "preregular-combination.maude", line 11 (fmod SIG):
    operator f has been imported from both
    "preregular-combination.maude", line 7 (fmod SIG2) and
    "preregular-combination.maude", line 8 (fmod SIG2) with
    no common ancestor.
Warning: sort declarations for operator f failed
    preregularity check.
```

As specifications grow complex, even more particularly in the case of a highly fine-grained specification such as CMSOS, with a large number of modules, sets and subsets relations that are directly mapped as sorts and subsorts, the possibility of making a module inclusion that results in a loss of preregularity is to be taken seriously. This is, unfortunately, a characteristic MEL and not MMT. What the tool currently lacks is a way of warning users about non-preregular modules. This is done by the Maude engine itself only after the compilation is done, which can be very confusing for a user that has little knowledge of the implementation of the tool.

Another improvement in Maude that would bring *Maude MSOS Tool* closer in functionality of Mosses's MSOS Tool is the ability to dynamically load modules. Currently, the loading of modules through Maude's '`in`' or '`load`' commands (they differ only in the verbosity of the loading process) is not part of the formalism itself (as the '`rewrite`' command its, for example). Even though MMT automatically includes modules that satisfy certain requirements (Section 4.2.1), they must already be loaded into Full Maude to actually be included. A consequence of this is that all modules must be manually loaded in a required order that corresponds to their dependency relation. A way of avoiding this laborious step is to dynamically load required modules, if they have not been loaded before. Mosses's MSOS Tool uses Prolog's '`ensure_loaded`' function to achieve this effect.

Finally, `LOOP-MODE`'s way of handling user input treats loaded files as an endless stream of tokens. In other words, files cannot be compilation units themselves. This is probably why it is common in Full Maude to use *delimited modules* such as '`msos ... sosm`', '`tmod ... endtm`', '`omod ... endom`', etc., as compilation units. This makes *Maude MSOS Tool* different from Mosses's MSOS Tool, which *files* are modules with MSDF constructions, named by their directory hierarchy and filename.

7.1.5 Automatic variables

The automatic typing of variables, based on their names obviously limit the name of variables that are allowed on MSDF specifications. This has an awkward effect on rules that needs several variables of the same type, such as the following, taken from Appendix E.1, where variables '`Int`' and '`Int'`' have the same name, but correspond to different objects: '`Int`' refers to the process id, while '`Int'`' is the semaphore value.

```
                          Int' == 0
 -- -----------------------------------------------------
 prc (Int, down) : Proc -{sem = Int', sem' = Int', -}->
```

```
prc (Int, down) .
```

This is is a matter of specification engineering: a possible approach on the rule above is to create another set, '`SemVal`' that is a superset of integers and use a variable name '`SemVal`' instead of '`Int'`'. This problem also does not appear in the CMSOS and Mini-Freja specifications.

Even though we will not attempt to prove it, we believe that a type inference algorithm would have serious problems in the face of an order-sorted specification and we hope that the problem should be made clear with the following example.

```
Exp .
Exp ::= Int | add(Exp, Exp) .

             A -{...}-> A'
--- ------------------------------
add(A,B) : Exp -{...}-> add(A',B) .

             D -{...}-> D'
--- ------------------------------
add(C,D) : Exp -{...}-> add(C,D') .

add(E,F) --> E + F .
```

A proper type inferencing algorithm would be able to determine the type of variables '`A`'–'`F`', which are, respectively, '`Exp`', '`Exp`', '`Int`', '`Exp`', '`Int`', and '`Int`'. The only two variables which the type might be safely inferred are '`E`' and '`F`', because '`_+_`' is only defined for '`Int`' in this case. Otherwise, due to the subset inclusion of '`Int`' into '`Exp`' it is not possible to know, for example that '`C`' should be '`Int`' and not '`Exp`'. A brute force approach of adding all possible combinations of '`Exp`' and '`Int`' to the other rules would work (tests show this), but it would generate an unnecessary number of spurious rewrites. The combination of subset inclusion and metavariables could be exponential on a large specification.

7.2 Enhancements to the tool — future work

The relationship between MSOS and MRS developed by [6, 44] uses conditional rewrite rules for the semantics of MSOS transitions, since they cover the general case of non-deterministic, non-terminating transition systems. However, for a deterministic, terminating, fragment of a transition system, one could use *equations* instead of rules and keep the later only in cases where non-determinism or non-termination might happen. For example, rule '`[let1]`' on page 42 could be converted to a conditional equation (with a "one step" conditional equality) such as:

```
ceq [let1] : { let X = I in E end, {(env = Env), PR} }
    = [ let X = I in E' end, {(env = Env), PR'} ]
```

```
if Env' := override (Env, X, I) /\
   [ E', {(env = Env'), PR'} ] := { E, {(env = Env'), PR} } .
```

Unfortunately, combining, on a single specification, equations and rules would probably generate difficulties. Consider MSDF rules for concurrent processes, variable assignment, and looping commands that would be converted to conditional rewrite rules since non-determinism or non-termination might occur. Usually, these rewrite rules use rewriting conditions such as:

```
crl { E1 || E2, R } => [ E'1 || E2, R' ]
 if { E1, R } => [ E'1, R' ] .
```

where '`_||_`' is an associative-commutative operator, a "soup" of expressions. There will be no equation that would match the condition (they are equalities, not rewrites). It should be remarked, that, in the case of a concurrent specification of a programming language, a common solution in Rewriting Logic adapted to the MRS case is to use a *multiset of configurations*, instead of a single configuration [48]. Each configuration would have its own program text and record; concurrent execution of the system would be given by rewrite rules over this multiset and each configuration itself would be rewritten using equations for the deterministic, terminating parts and rules for the non-deterministic, non-terminating parts. This may open the possibility of adding *true concurrency* to MSDF specifications, instead of the current interleaving semantics. This would probably require also some work on new label categories to support a true concurrent trace of execution in face of MSOS requirements of composability.

Another limitation is that model checking of MSDF specifications with conditional rewrites representing the transition rules' premises is problematic since rewrites in the conditions are assumed "scratch pad rewrites" in rewriting logic [40]. Thus, states that exist only in the conditions cannot be specified in the query (LTL formula) to the model checker. For example, on the main transitions, the bindings environment is always its initial value, while the actual bindings to identifiers happen in the conditions. With this limitation queries to the model checker must be made by observing changes to mutable components, such as the store, or by exploiting some property that involves the entire program text, and not some part. Also, conditional rewrites often slow down the rewriting process [63]. A possible solution to these is to generate only *unconditional rewrites*, following the ideas of *evaluation contexts* [21] and the use of *rewriting strategies* [14] to replace the need for the '`step`' rule (Section 2.4).

The combination of evaluation contexts and unconditional equations might lead to a significant increase in performance and (we believe) at the penalty of a significant decrease in readability, according to a prototype we developed using this technique for a subset of the Concurrent ML language (`http://www.ic.uff.br/~cbraga/losd/specs/cml-cps/cml.maude`) as seen in [11] and [48] shows—a further incentive to add these capabilities to MMT.

Another possible source of enhancement might come inspired by ideas in "partial evaluation" results. The idea is that it is quite possible that a given program P does not use the full set of transitions rules present on a specification $\mathcal{S}$. One could then generate a "specialized specification" $\mathcal{S}_{\mathsf{P}}$ that contains only the necessary constructors,

rules, and equations for the execution and verification of the program P. If the set of rules omitted is large, the matching of terms to rules will probably be significantly faster in the Maude engine. When Maude acquires a compiler, one could effectively compile a program P creating the specialized specification $\mathcal{S}_{\mathsf{P}}$ and then compiling this specification using Maude's compiler.

The tool needs better user-interface support. Currently, errors may happen at three different levels: those that MMT reports, those that MMT fails to report but Full Maude detects, and those that are left to the Maude rewrite engine to warn about. This is confusing, but a more robust solution would need to endow MMT with significant knowledge of MEL in order to foresee any problems a particular module might have after compilation. In the same way that Maude's '`metaParse`' command may return an error indicating a source of parsing problems, there could be other meta-level commands that analyze metamodules and report any problems in the same way.

Finally, following the example from LETOS (Chapter 3), a LaTeX typeseting capability and some form of trace execution that is closer to the MSOS domain than to the Maude domain would greatly influence on the usability of the tool.

7.3 Contributions

The main contributions of our work are: (i) developing an MSOS interpreter that uses a specification language that is closer to the domain of MSOS specifications than to Maude specifications. This fact opens the possibility of the automatic typeseting of MSOS specifications that are close to graphical MSOS (and SOS) notations; (ii) implementing a new conversion from MSOS to Rewriting Logic, based on the subsequent work of Braga and Meseguer in [8, 44]; (iii) demonstrating the usability of the tool and the CMSOS framework by developing different programming languages specifications; (iv) demonstrating what can be accomplished when one develops a formal tool in the Maude environment, since it allows the use of other formal tools already available with MSDF specifications. We have demonstrated this by simulating and model checking concurrent programs and distributed algorithms; (v) providing an example of a non-trivial extension of Full Maude; (vi) finally, one of the objectives of the tool, that is, integrating several formal tools by using Maude as aggregating technology has recently put to test by extending MMT with Verdejo's Strategy Language for Maude [45] by Braga[1] and using the combined tool to give the semantics of core GPH (Glasgow Parallel Haskell) [3].

[1]Personal communication, Feb. 2005.

Bibliography

[1] Luca Aceto, Willem Jan Fokkink, and Chris Verhoef. Conservative extension in structural operational semantics. Research Series RS-99-24, BRICS, Department of Computer Science, Institute of Electronic Systems, Aalborg University, September 1999. 23 pp. Appears in the *Bulletin of the European Association for Theoretical Computer Science*, 70:110–132, 1999.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java: Basic Techniques.* Cambridge University Press, Cambridge, UK, February 1997.

[3] Clem Baker-Finch, David J. King, and Phil Trinder. An operational semantics for parallel lazy evaluation. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 162–173, New York, NY, USA, 2000. ACM Press.

[4] G. Berry and G. Boudol. The chemical abstract machine. In *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*, pages 81–94. ACM Press, New York, 1990.

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

[6] Christiano Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics.* PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, September 2001. `http://www.ic.uff.br/~cbraga`.

[7] Christiano Braga, Hermann Haeusler, José Meseguer, and Peter Mosses. Maude Action Tool: Using reflection to map action semantics to rewriting logic. In Teodor Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 407–421. Springer-Verlag, 2000.

[8] Christiano Braga and José Meseguer. Modular rewriting semantics in practice. *Electronic Notes in Theoretical Computer Science*, 117:393–416, 2005.

[9] Walter S. Brainerd and Lawrence H. Landweber. *Theory of Computation.* John Wiley and Sons, New York, 1974.

[10] Roberto Bruni and José Meseguer. Generalized rewrite theories. In *Thirtieth International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[11] Fabricio Chalub and Christiano Braga. A Modular Rewriting Semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, July 2004. `http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics`.

[12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude as a metalanguage. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 237–250. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[13] Manuel Clavel, Francisco Durán, Steven Eker, Narciso Martí-Oliet, Patrick Lincoln, José Meseguer, and José Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, `http://maude.csl.sri.com`, January 1999.

[14] Manuel Clavel, Francisco Durán, Steven Eker, Narciso Martí-Oliet, Patrick Lincoln, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.1)*. SRI International and University of Illinois at Urbana-Champaign, `http://maude.cs.uiuc.edu`, March 2004.

[15] Manuel Clavel, Francisco Durán, Steven Eker, José Meseguer, and Mark-Oliver Stehr. Maude as a formal meta-tool. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20–24, 1999 Proceedings, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer-Verlag, 1999.

[16] Manuel Clavel, José Meseguer, and Miguel Palomino. Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic. In Fabio Gadducci and Ugo Montanari, editors, *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[17] Grit Denker, José Meseguer, and Carolyn Talcott. Protocol specification and analysis in Maude. In *Heintze, N. and Wing, J., editors, Proc. of Workshop on Formal Methods and Security Protocols*, June 1998. Indianápolis, Indiana.

[18] Grit Denker, José Meseguer, and Carolyn Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DISCEX 2000, Proc. Darpa Information Survivability Conference and Exposition, Hilton Head, South Carolina*, volume 1, pages 251–265. IEEE Computer Society Press, January 2000.

[19] Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, April 2003.

[20] Azade Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron A. Peled, editors, *CAV*, Lecture Notes in Computer Science. Springer, 2004.

[21] Matthias Felleisen. *The Calculi of λ-ν-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[22] Kokichi Futatsugi and R. Diaconescu. Cafeobj report. *World Scientific, AMAST Series*, 1998.

[23] J. Goguen, C. Kirchner, A. Megrelis, J. Meseguer, and T. Winkler. An introduction to obj3. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems, 1st International workshop, Orsay, France*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, New York, NY, July 1987.

[24] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[25] Joseph A. Goguen and Grant Malcolm. More higher order programming in OBJ3. In Joseph A. Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, chapter 9, pages 397–408. Kluwer, Boston, 2000.

[26] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[27] P. H. Hartel. LETOS – A lightweight execution tool for operational semantics. *Software—Practice and Experience*, 29(15):1379–1416, Sep 1999.

[28] A. E. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. *Lecture Notes in Computer Science*, 1101:132–147, 1996.

[29] Matthew Hennessy. *A Semantics of Programming Languages: An Elementary Introduction Using Operational Semantics*. John Wiley and Sons, 1990. Currently out of print; available from `http://www.cogs.susx.ac.uk/users/matthewh/semnotes.ps.gz`.

[30] Jørgen Iversen. *Formalisms and tools supporting Constructive Action Semantics*. PhD thesis, Univ. of Aarhus, 2005.

[31] Jørgen Iversen and Peter D. Mosses. Constructive action semantics for core ML. *IEE Proceedings*, 152(2), April 2005. Special issue on Language definitions and tool generation.

[32] S. Peyton Jones, editor. *Haskell 98 Language and Libraries; The Revised Report*. Cambridge University Press, 2003.

[33] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[34] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.

[35] Donald E. Knuth. *Selected Papers on Computer Languages.* CSLI Publications, Stanford, CA, USA, 2002.

[36] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[37] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, editor, *Handbook of Philosophical Logic, Second Edition, Volume 6*, Kluwer Academic Publishers, 2001. `http://maude.csl.sri.com/papers`.

[38] Narciso Martí-Oliet and José Meseguer. *Handbook of Philosophical Logic*, volume 61, chapter Rewriting Logic as a Logical and Semantic Framework. Kluwer Academic Publishers, second edition, 2001. `http://maude.cs.uiuc.edu/papers`.

[39] José Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02R, SRI International, Computer Science Laboratory, February 1990. Revised June 1990. Appendices on functorial semantics have not been published elsewhere.

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

[41] José Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, editor, *WADT'97*, volume 1376, pages 18–61. Springer, 1998.

[42] José Meseguer. Software specification and verification in rewriting logic. Lectures given at the NATO Advanced Study Institute International Summer School, Marktoberdorf, Germany, 2002. Available from `http://maude.cs.uiuc.edu`, 2003.

[43] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. Manuscript. `http://maude.cs.uiuc.edu/papers`.

[44] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *In Algebraic Methodology and Software Technology: proceedings of the 10th International Conference, AMAST 2004*, volume 3116 of *LNCS*, pages 364–378, Stirling, Scotland, UK, July 2004. Springer. ISSN 0302-9743, ISBN 3-540-22381-9.

[45] José Meseguer, Narciso Martí-Oliet, and Alberto Verdejo. Towards a strategy language for Maude. In Narciso Martí-Oliet, editor, *Proceedings of 5th International Workshop on Rewriting Logic and its Applications, WRLA 2004*, volume 117 of *Eletronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2005.

[46] José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Notes on model checking and abstraction in rewriting logic. http://maude.cs.uiuc.edu/.

[47] José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Equational abstractions. In Franz Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[48] José Meseguer and Grigore Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. Basin and M. Rusinowitch, editors, *Proceedings of the 2nd International Joint Conference on Automated Reasoning, IJCAR'04, (Cork, Ireland)*, volume 3097 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2004.

[49] Robert Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (Revised).* MIT Press, 1997.

[50] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[51] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[52] Peter D. Mosses. Fundamental Concepts and Formal Semantics of Programming Languages—an introductory course. Lecture notes, available at `http://www.daimi.au.dk/jwig-cnn/dSem/`, 2004.

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

[54] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing. John Wiley & Sons, Chichester, England, 1992.

[55] Peter Csaba Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic.* PhD thesis, University of Bergen, Norway, 2000. `http://maude.csl.sri.com/papers`.

[56] Mikael Pettersson. *Compiling Natural Semantics*, volume 1549 of *Lecture Notes in Computer Science*. Springer, 1999.

[57] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

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

[59] Alexandre Rademaker, Christiano Braga, and Alexandre Sztajnberg. A rewriting semantics for a software architecture description language. 2005. To appear.

[60] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*, pages 289–297, Snowbird, Utah, July 1988.

[61] John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.

[62] John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, Cambridge, England, 1998.

[63] Grigore Rosu. From conditional to unconditional rewriting. In *Proc. 17th Int. Workshop on Algebraic Development Techniques (WADT 2004)*, 2004.

[64] Mark-Oliver Stehr and Ambarish Sridharanarayanan. Formal specification of sectrace. In *Workshop on Context Sensitive Systems Assurance (Contessa'03)*, 2003.

[65] Lars Thorup and Mads Tofte. Object-oriented programming and Standard ML. In John H. Reppy, editor, *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Orlando, Florida*, number 2265 in Rapport de recherche, pages 41–49. INRIA, June 1994.

[66] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In J. Jouannaud, editor, *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architectures, Nancy, France*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, New York, NY, September 1985.

[67] David Ungar and Craig Chambers. Self: The power of simplicity. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, number 12, pages 227–242. ACM, 1987.

[68] José Alberto Verdejo. *Maude como um marco semântico ejecutable*. PhD thesis, Universidad Complutense Madrid, 2003.

[69] Patrick Viry. Equational rules for rewriting logic. *Theor. Comput. Sci.*, 285(2):487–517, 2002.

APPENDIX A – Constructive MSOS

In this Appendix, we describe the remainder of the CMSOS constructions that were omitted from Section 6.1.

A.1 Expressions

We begin by describing the modules for the application of operators and application of identifiers. Rule '`app-op1`' evaluates the argument of the application—additional rules are needed for a particular case. As we mentioned earlier, rule '`app-op2`' is a convenience to the user and is used in the particular case that the original argument is a tuple of values, and there is an (externally defined) equation that gives the result of the application of operation '`Op`' to the values in '`Value*`'.

```
msos Cons/Exp/app-Op is
 Exp ::= app Op Arg .
 Value ::= tup Value* .

                          Arg -{...}-> Arg'
 [app-op1] -- -------------------------------------
           (app Op Arg) : Exp -{...}-> app Op Arg' .

                Value' := apply-op (Op, Value*)
 [app-op2] -- ------------------------------------
           (app Op (tup Value*)) : Exp --> Value' .
sosm
```

The '`app-Id`' module defines the application of an identifier to an argument. The actual rules that will govern each particular case of identifiers and arguments will be defined in subsequent modules. It is usually the case where the identifier will evaluate, by looking up on the environment, to an *operator*, and then the associated operation will be applied.

```
msos Cons/Exp/app-Id is
 Exp ::= app Id Arg .
```

```
                Id --> Id'
 -- --------------------------------
 (app Id Arg) : Exp --> app Id' Arg .

               Arg -{...}-> Arg'
 -- -------------------------------------
 (app Id Arg) : Exp -{...}-> app Id Arg' .
sosm
```

A.2 Declarations

Continuing, the module '`simult-seq`' defines the simultaneous creation of a set of bindings. This means that the bindings are created independently of each other. This is a restricted variant of the '`simult`' construction (not shown here) that evaluates the bindings sequentially, while the general version evaluates the bindings in any order. The bindings are evaluated and then joined together by the '`+++`' operator at the end.

```
msos Cons/Dec/simult-seq is
 Dec ::= simult-seq Dec Dec .

                        Dec1 -{...}-> Dec'1
 -- ---------------------------------------------------------
 (simult-seq Dec1 Dec2) : Dec -{...}-> simult-seq Dec'1 Dec2 .

                        Dec2 -{...}-> Dec'2
 -- ----------------------------------------------------------
 (simult-seq Env1 Dec2) : Dec -{...}-> simult-seq Env1 Dec'2  .

        Env := (Env1 +++ Env2)
 -- -----------------------------------
 (simult-seq Env1 Env2) : Dec --> Env .
sosm
```

The construction '`accum`' defines a cumulative declaration, a counterpart of the '`simult-seq`' construction, where bindings have available to themselves all bindings generated so far.

```
msos Cons/Dec/accum is
 Dec ::= accum Dec Dec .

 Label = {env : Env,...} .

                   Dec1 -{...}-> Dec'1
 -- ----------------------------------------------
 (accum Dec1 Dec2) : Dec -{...}-> accum Dec'1 Dec2 .

    Env' := (Env1 / Env0),
```

```
                    Dec2 -{env = Env', ...}-> Dec'2
 -- ------------------------------------------------------------
 (accum Env1 Dec2) : Dec -{env = Env0, ...}-> accum Env1 Dec'2 .

         Env := (Env2 / Env1)
 -- ------------------------------
 (accum Env1 Env2) : Dec --> Env .
sosm
```

The '`Exp/local`' construction takes a declaration and an expression, and evaluates the expression in terms of the declarations. The declaration is first evaluated into '`Env`', a set of bindings that are used to override the current environment, '`Env0`', that surrounds the construction. These newly created bindings, '`Env'`', are used to evaluate each step of the expression '`Exp`'. When this expression reaches a final value '`Value`', the whole construction evaluates to this value.

```
msos Cons/Exp/local is
 Exp ::= local Dec Exp .

 Label = { env : Env, ...} .

                   Dec -{...}-> Dec'
 -- -------------------------------------------
 (local Dec Exp) : Exp -{...}-> local Dec' Exp .

    Env' := (Env / Env0),
                   Exp -{env = Env', ...}-> Exp'
 -- --------------------------------------------------------
 (local Env Exp) : Exp -{env = Env0, ...}-> local Env Exp' .

 (local Env Value) : Exp --> Value .
sosm
```

A.3 Commands

Module '`seq-Cmd-Exp`' defines a construction '`seq`' which evaluates the sequence of commands given as its first argument and then evaluates the expression, returning its value. It may be regarded as the body of an imperative language function where the last command is a '`return`', that returns the value of '`Exp`'.

```
msos Cons/Exp/seq-Cmd-Exp is
 Exp ::= seq Cmd Exp .

                 Cmd -{...}-> Cmd'
 -- ----------------------------------------
 (seq Cmd Exp) : Exp -{...}-> seq Cmd' Exp .
```

```
 (seq skip Exp) : Exp --> Exp .
sosm
```

The construction defined by the module ‘seq-Exp-Cmd’ is equivalent the one defined in ‘seq-Exp-Cmd’ but the expression is evaluated before the commands, but its evaluated value is returned after the evaluation of the list of commands given as the second argument.

```
msos Cons/Exp/seq-Exp-Cmd is
 Exp ::= seq Exp Cmd .

                 Exp -{...}-> Exp'
 -- ---------------------------------------
 (seq Exp Cmd) : Exp -{...}-> seq Exp' Cmd .

                   Cmd -{...}-> Cmd'
 -- -------------------------------------------
 (seq Value Cmd) : Exp -{...}-> seq Value Cmd' .

 (seq Value skip) : Exp --> Value .
sosm
```

The ‘cond’ construction is a command conditional, that selects which command to execute, depending on the evaluation of its first argument.

```
msos Cons/Cmd/cond is
 Cmd ::= cond Exp Cmd Cmd .
 Value ::= Boolean .

                        Exp -{...}-> Exp'
 -- ----------------------------------------------------
 (cond Exp Cmd1 Cmd2) : Cmd -{...}-> cond Exp' Cmd1 Cmd2 .

 (cond tt Cmd1 Cmd2) : Cmd --> Cmd1 .

 (cond ff Cmd1 Cmd2) : Cmd --> Cmd2 .
sosm
```

The ‘while’ construction is the traditional looping construction, whose meaning is based on the command sequencing and conditional commands.

```
msos Cons/Cmd/while is
 see Cons/Cmd/cond, Cons/Cmd/seq .

 Cmd ::= while Exp Cmd .
```

```
 (while Exp Cmd) : Cmd -->
   cond Exp (seq Cmd (while Exp Cmd)) skip .
sosm
```

Now we enter the set of constructions that deal with mutable information, modelled through ‘`Store`’, a read-write component indexed by ‘`st`’ and ‘`st'`’. In order to achieve the desired generality, CMSOS uses the set ‘`Var`’ to represent the variables that are bound to memory locations (represented by the set ‘`Cell`’).

```
msos Cons/Var is
 Var .
 Var ::= Cell .
sosm
```

The evaluation of a variable as an expression expects it to evaluate to a ‘`Cell`’. Once this happens, it value bound to the cell in the store is returned.

```
msos Cons/Exp/Var is
 Exp ::= Var .
 Label = {store : Store, store' : Store, ...} .

    Var -{...}-> Var'
 -- ----------------------
 Var : Exp -{...}-> Var' .

               Value := lookup(Cell, Store)
 -- ----------------------------------------------------
 Cell : Exp -{store = Store, store' = Store,-}-> Value .
sosm
```

Variables may or may not be related to identifiers. If they are, then we use the following module, which defines the evaluation of identifiers in the context of variables. It is expected that an identifier in this case evaluates to a cell. To appear in environments, a cell must be a ‘`Bindable`’ value.

```
msos Cons/Var/Id is
 Var ::= Id .
 Bindable ::= Cell .

 Label = {env : Env, ...} .

      Cell := lookup (Id, Env)
 -- ----------------------------
 Id : Var -{env = Env,-}-> Cell .
sosm
```

The construction ‘assign-seq’ changes the value pointed by ‘Var’ to the value obtained by the evaluation of ‘Exp’.

```
msos Cons/Exp/assign-seq is
 Exp ::= assign-seq Var Exp .

 Label = {store : Store, store' : Store, ...} .

                        Var -{...}-> Var'
 -- -------------------------------------------------------
 (assign-seq Var Exp) : Exp -{...}-> assign-seq Var' Exp .

                         Exp -{...}-> Exp'
 -- ---------------------------------------------------------
 (assign-seq Cell Exp) : Exp -{...}-> assign-seq Cell Exp' .

 def lookup(Cell, Store),
             Store' :=  (Cell |-> Storable) / Store
 -- ------------------------------------------------
 (assign-seq Cell Storable) : Exp
   -{store  = Store, store' = Store',-}-> Storable .
sosm
```

These constructions are complements of each other. The first ‘ref’ evaluates the variable passed as it sole argument. The second ‘deref’ evaluates the expression as its sole argument and expects it to be evaluated to a ‘ref Cell’. These constructions are used to bring the imperative facet to a functional language by creating a special “value,” ‘ref’ that holds a pointer. On a purely imperative language, these constructions are not necessarily needed, as variables may be used directly. They work as follows: if we want to bind an identifier i to some value v through the store, we first create a new cell c on the store that is mapped to v and enclose this cell by the ‘ref’ construction. The identifier is now bound indirectly to v. To access this value on a subsequence computation, we will call ‘deref i’, which will evaluate to a ‘deref (ref (c))’. The evaluation of this last expression, according to the rules from the ‘Cons/Exp/Var’ module, will then evaluate to the desired value v.

```
msos Cons/Exp/ref is
 Exp ::= ref Var .
 Value ::= ref Cell .

             Var -{...}-> Var'
 -- -------------------------------
 (ref Var) : Exp -{...}-> ref Var' .
sosm

msos Cons/Var/deref is
 Var ::= deref Exp .
```

```
 Value ::= ref Cell .

              Exp -{...}-> Exp'
 -- ------------------------------------
 (deref Exp) : Var -{...}-> deref Exp' .

 (deref (ref Cell)) : Var --> Cell .
sosm
```

The following construction returns the value bound through the variable '`Var`'.

```
msos Cons/Exp/assigned is
 Exp ::= assigned Var .
 Label = {store : Store, store' : Store, ...} .
                 Var -{...}-> Var'
 -- ------------------------------------------
 (assigned Var) : Exp -{...}-> assigned Var' .

    Storable := lookup (Cell ,Store)
 -- ---------------------------------------------------
 (assigned Cell) : Exp -{store = Store,
                         store' = Store,-}-> Storable .
sosm
```

The '`alloc`' construction creates a new entry on the store for the value obtained from the evaluation of '`Exp`' and then returns a pointer to this entry.

```
msos Cons/Var/alloc is
 Var ::= alloc Exp .

 Label = {store : Store, store' : Store, ...} .

              Exp -{...}-> Exp'
 -- ------------------------------------
 (alloc Exp) : Var -{...}-> alloc Exp' .

    Cell := new-cell (Store),
        Store' := (Cell |-> Storable) / Store
 -- -----------------------------------------
 (alloc Storable) : Var -{store = Store,
                          store' = Store',-}-> Cell .
sosm
```

A.4 Abstractions

Continuing the semantic of abstractions from Section 6.1, the '`Cons/Dec/app`' module defines the abstract syntax for the application of an argument to a parameter. This

application will be converted into a declaration in which the argument will be bound to the parameter. This construction will be used on the application of closures to expressions.

```
msos Cons/Dec/app is
 Dec ::= app Par Arg .

                 Arg -{...}-> Arg'
 -- -----------------------------------------
 (app Par Arg) : Dec -{...}-> (app Par Arg') .
sosm
```

Module 'Cons/Par/bind' defines the actual parameters of abstractions and how they become declarations when applied to arguments. The simplest form of parameter is the 'bind', which takes a single identifier. When applied to a bindable value, it is converted into a binding.

```
msos Cons/Par/bind is
 see Cons/Dec/app .

 Par ::= bind Id .

 (app (bind Id) Bindable) : Dec --> (Id |-> Bindable) .
sosm
```

In order to bind several parameters simultaneously, the following module should be used, which defines the 'tup' parameter construction.

```
msos Cons/Par/tup is
 see Cons/Dec/app, Cons/Dec/simult .
 see Cons/Exp/tup .

 Par ::= tup Par* .

 (app (tup(Par, Par*)) (tup (Bindable, Bindable*))) : Dec -->
    (simult (app Par Bindable)
            (app (tup(Par*)) (tup(Bindable*)))) .

 (app tup() tup()) : Dec --> void .
sosm
```

For recursive bindings we use the concept of *finite unfolding*, with reclosures. We define a recursive binding with the construction 'rec' applied before any declaration. This creates a special type of declaration where the first argument is always 'rec Dec'. As this declaration is evaluated to generate bindings, 'rec Dec' is evaluated over and over again, having the effect of an finite unfolding.

```
msos Cons/Dec/rec is
 see Cons/Dec/bind, Cons/Dec/simult,
     Cons/Dec/simult-seq, Cons/Exp/close,
     Cons/Abs/closure .

 Dec ::= rec Dec .
 Dec ::= reclose Dec Dec .

 (rec Dec) : Dec --> (reclose (rec Dec) Dec) .

 (reclose (rec Dec) (bind Id (close Abs))) : Dec -->
  (bind Id (close (closure (rec Dec) Abs))) .

 (reclose (rec Dec) (simult-seq Dec1 Dec2)) : Dec -->
 (simult-seq (reclose (rec Dec) Dec1) (reclose (rec Dec) Dec2)) .

 (reclose (rec Dec) (simult Dec1 Dec2)) : Dec -->
 (simult (reclose (rec Dec) Dec1) (reclose (rec Dec) Dec2)) .
sosm
```

A.5 Concurrency

The construction ‘`start`’ signals the creation of a new thread. It is expected that the body of the thread consists of an abstraction that will be applied to the empty tuple upon activation. The module ‘`Cons/Cmd/start`’ defines its meaning: after the evaluation of the expression ‘`Exp`’, the construction signalizes the creation of a new thread by “producing” the abstraction in the write-only component ‘`starting`’.

```
msos Cons/Cmd/start is
 Cmd ::= start Exp .

 Label = {starting' : Abs*, ...} .

 Value ::= Abs .

              Exp -{...}-> Exp'
 -- --------------------------------------
 (start Exp) : Cmd -{...}-> (start Exp') .

 (start Abs) : Cmd -{starting' = Abs,-}-> skip .
sosm
```

Systems are composed of commands. The following module describes the evaluation of a command in the context of a system. If during the execution a command a new thread is detected on the ‘`starting'`’ component, it is removed from that component and put into the pool of running threads, bound together by the ‘`conc`’ construction. If

no thread is signalized, then the execution continues as usual. As threads end, i.e., they evaluate to the 'skip' command, they are removed from the pool.

```
msos Cons/Sys/Cmd is
 see Cons/Sys/conc, Cons/Cmd/effect,
     Cons/Exp/Abs,
     Cons/Exp/app, Cons/Exp/tup .

 Sys ::= Cmd | skip .

 Label = {starting' : Abs*, ...} .

      Cmd -{starting' = Abs, ...}-> Cmd'
-- --------------------------------------
Cmd : Sys -{starting' = (), ...}->
     conc Cmd' effect (app Abs tup()) .

     Cmd -{starting' = (), ...}-> Cmd'
 -- --------------------------------------
 Cmd : Sys -{starting' = (), ...}-> Cmd' .

 (conc skip Sys) : Sys --> Sys .

 (conc Sys skip) : Sys --> Sys .
sosm
```

The following modules deal with synchronous message-passing. Let us begin with the creation of channels, represented by the set 'Chan'. Each channel has an unique identifier an integer.

```
msos Data/Chan is
 Chan .
 Chan ::= chan Int .
sosm
```

The 'alloc-chan' creates a new channel to be used and adds it to the read-write 'Chans' components, that keeps track of all channels created so far. The function 'new-chan' is defined externally through equations to simplify the specification, it creates a new channel by looking at the set of channels ('Chans') and returning an unused identification.

```
msos Data/Chans is
 Chans .
 Chans = (Chan) Set .
sosm

msos Cons/Exp/alloc-chan is
```

```
 Exp ::= alloc-chan .

 Label = {chans : Chans, chans' : Chans, ...} .

 Value ::= Chan .
 Chan ::= new-chan (Chans) .

 Chan := new-chan (Chans), Chans' := Chans + { Chan }
 -- --------------------------------------------------------------
 alloc-chan : Exp -{chans = Chans, chans' = Chans', -}-> Chan .
sosm
```

In order to model the synchronous message passing of values, CMSOS follows the ideas of Concurrent ML. The write-only component '`event`' models the production of events during a computation. Threads block after producing events, waiting for other threads to produce matching events. Concurrent ML defines several different events and their matching relationships, but in this case the only matching events are the '`sending`' and '`receiving`', which model the sending and receiving of values through a particular channel.

The '`send-chan-seq`' function receives two expressions as arguments: the first is evaluated into a channel identification and the second is evaluated to the value that should be sent to that channel. After both expressions are evaluated, it produces the event '`sending`' with the channel and value.

```
msos Cons/Cmd/send-chan-seq is
 Cmd ::= send-chan-seq Exp Exp .

 Label = {event' : Event*, ...} .

 Event ::= sending Chan Value .

 Value ::= Chan .

              Exp1 -{...}-> Exp1'
 -- -------------------------------------------
 (send-chan-seq Exp1 Exp2) : Cmd -{...}->
                   (send-chan-seq Exp1' Exp2) .

              Exp2 -{...}-> Exp2'
 -- -------------------------------------------
 (send-chan-seq Chan Exp2) : Cmd -{...}->
                   (send-chan-seq Chan Exp2') .

 (send-chan-seq Chan Value) : Cmd
    -{event' = (sending Chan Value),-}-> skip .
sosm
```

The construction 'recv-chan' receives a value through the channel that is obtained from the evaluation of the expression 'Exp'. Here we deviate from the original MSDF specification since that used variable unification, a feature not available in Maude as of version 2.1.1. Originally, the function 'recv-chan' also used a free variable 'Value' that unifies when there is a match against a 'sending' event. Our solution is to, after evaluating 'Exp' to a 'Chan', we put a *placeholder* 'ph Chan' in place of the free variable. When the matching occurs we update this placeholder with the correct value, following the technique we applied while defining a Modular Rewriting Semantics of Concurrent ML in [11]

```
msos Cons/Exp/recv-chan is
 Exp ::= recv-chan Exp .

 Label = {event' : Event*, ...} .

 Event ::= receiving Chan .

 Value ::= ph Chan | Chan .

                  Exp -{...}-> Exp'
 -- ---------------------------------------------
 (recv-chan Exp) : Exp -{...}-> (recv-chan Exp') .

 (recv-chan Chan) : Exp -{event' = (receiving Chan),-}->
                               (ph Chan) .
sosm
```

Here is the module that describes the matching of events: if a 'sending' and 'receiving' event are produced by any two threads, they synchronize and the value is passed from one thread to another. This is make using a *metafunction* 'update-ph' that updates the placeholder with the transmitted value. This metafunction iterates over the program text at the metalevel to make the substitution, and for this reason our solution does not depend on the signature of the language and hence does not harm the modularity of the specification.

```
msos Cons/Sys/conc-chan is
 Sys ::= conc Sys Sys .

 Sys ::= update-ph (Sys, Chan, Value) .

 Label = {event' : Event*, ...} .

 Event ::= sending Chan Value
        | receiving Chan .

 Sys1 -{event' = (sending Chan Value),-}-> Sys1',
       Sys2 -{event' = (receiving Chan),-}-> Sys2'
```

```
 -- -----------------------------------------------
 (conc Sys1 Sys2) : Sys -{event' = (),-}->
      (conc Sys1' update-ph (Sys2', Chan, Value)) .

 Sys2 -{event' = (sending Chan Value),-}-> Sys2',
         Sys1 -{event' = (receiving Chan),-}-> Sys1'
 -- ------------------------------------------------
 (conc Sys1 Sys2) : Sys -{event' = (),-}->
       (conc update-ph (Sys1', Chan, Value) Sys2') .
sosm
```

Finally, we must forbid threads to evaluate if they contain unmatched events. This is achieved by enclosing the entire system with the '`quiet`' construction. This construction only let the system evolve when all events have been matched. When this happen the '`event`' component is always the empty sequence '`()`'.

```
msos Cons/Sys/quiet is
 see Cons/Sys/Cmd .

 Sys ::= quiet Sys .

 Label = {event' : Event*, ...} .

               Sys -{event' = (), ...}-> Sys'
 -- -------------------------------------------
 (quiet Sys) : Sys -{event' = (), ...}-> (quiet Sys') .

 (quiet skip) : Sys --> skip .
sosm
```

APPENDIX B – ML specification

This Chapter contains the complete ML specification that was described in Section 6.1.2.

B.1 Expressions

We begin by describing ML expressions and their CMSOS counterparts. First we need to gather all CMSOS modules that are necessary for the definitions of expressions in ML. This is done by creating a module 'Lang/ML/Exp' as follows. The module contains explicit references to all CMSOS constructions needed. It also defines that the set of values ('Value') and operators ('Op' are "bindable" in environments, that the set of values are "passable" to procedural abstractions, and that the set of operators contains the constants 'plus', 'times', etc.

```
msos Lang/ML/Exp is
 see Cons/Prog, Cons/Prog/Exp .

 see Cons/Exp, Cons/Exp/Boolean, Cons/Exp/Int,
     Cons/Exp/Id, Cons/Exp/cond, Cons/Exp/app-Op,
     Cons/Exp/app-Id, Cons/Exp/tup, Cons/Exp/tup-seq .

 see Cons/Arg, Cons/Arg/Exp .

 see Cons/Op .

 see Cons/Id .

 Bindable ::= Value | Op .

 Op ::= plus | times | minus | eq | lt | gt .

 Passable ::= Value .
sosm
```

The following module contains the initial dynamic basis for ML expressions. It is defined as a system module that includes the MSDF module 'Lang/ML/Exp'. Recall that we must define the operation 'apply-op' externally and this is done here for each 'Op'

constant declared on the 'Lang/ML/Exp' module. Following, we create the initial environment with the default associations of identifiers to operators. We reuse the names of the operator as identifiers, created with the coercion function 'ide' on the mapping.

```
mod Lang/ML/Exp' is
 including Lang/ML/Exp .
 including QID .

 vars i1 i2 : Int .

 eq apply-op (plus, (i1, i2)) = i1 + i2 .
 eq apply-op (minus, (i1, i2)) = i1 - i2 .
 eq apply-op (times, (i1, i2)) = i1 * i2 .
 eq apply-op (eq, (i1, i2)) = if i1 == i2 then tt else ff fi .
 eq apply-op (lt, (i1, i2)) = if i1 < i2 then tt else ff fi .
 eq apply-op (gt, (i1, i2)) = if i1 > i2 then tt else ff fi .

 op ide : Qid -> Id .
 op ide : Op -> Id .
 op op : Qid -> Op .

 eq init-env = (ide(eq) |-> eq +++ ide(lt) |-> lt +++
                ide(gt) |-> gt +++ ide(plus)  |-> plus +++
                ide(times) |-> times +++ ide(minus) |-> minus) .


 eq op ('+) = plus .   eq op ('*) = times .
 eq op ('-) = minus .  eq op ('<) = lt .
 eq op ('>) = gt .     eq op ('=) = eq .
endm
```

We begin the grammar by specifying "special constants" (⟨ scon ⟩), which are currently only integers, represented by the non-terminal 'integer literal'.

► *Special constants*

⟨ scon ⟩ ⟶ ⟨ integer literal ⟩

Special constants are converted verbatim to CMSOS constructions.

► *Infix operators*

Next, the rules for "infix operators" (⟨ inop ⟩) with are the operators that appear infixed in expressions. They can be either the equals sign or some symbol as defined by the non-terminal ⟨ symbolic id ⟩. We expect this non-terminal to be provided by the lexical analysis. Both are converted to an 'Op' in CMSOS through the coercion function 'op' that has as argument a quoted-identifier.

⟨ inop ⟩ → ‘=’ | ⟨ symbolic id ⟩

Let s range over ⟨ symbolic id ⟩.

```
⟦=⟧  = op('=)
⟦s⟧  = op(s)
```

► *Identifiers*

Identifiers defined by the nonterminal ⟨ id ⟩ (also provided by the lexical analysis phase) are converted into ‘`Id`’s using the coercion function ‘`ide`’ in the same way as the ‘`op`’ function.

⟨ vid ⟩ → ⟨ id ⟩

Let $\mathtt{i}$ range over ⟨ vid ⟩.

```
⟦i⟧  = ide(i)
```

► *Atomic expressions*

Being an abstract syntax version of the ML syntax, we avoid the use of different non-terminals to represent atomic, application, infix and complete expressions. However we opted to use that subdivision for ease of presentation of the specification. These are the rules for atomic expressions, which are special constants, identifiers, and tuple of expressions.

⟨ exp ⟩ → ⟨ scon ⟩ | ⟨ vid ⟩ | ‘()’ | ‘(’ ⟨ exp ⟩ ‘)’ | ‘(’ ⟨ exp ⟩* ‘)’

Let $\mathtt{e}_{\mathtt{i}}$ range over ⟨ exp ⟩.

```
⟦()⟧     = tup()
⟦(e)⟧    = ⟦e⟧
⟦(e ∗ )⟧ = tup-seq(⟦e∗⟧)
```

► *Application expressions*

Application expressions are used to invoke a procedural abstraction and are converted to the CMSOS construction ‘`app`’.

⟨ exp ⟩ → ⟨ exp ⟩ ⟨ exp ⟩

$[\![e_0\ e_1]\!]$ = app $[\![e_0]\!]$ $[\![e_1]\!]$

► *Infix expressions*

Infix expressions contains the infix operators, which are converted to the application of the operator receiving an argument the sequential tuple formed by both expressions.

⟨ exp ⟩ → ⟨ exp ⟩ ⟨ inop ⟩ ⟨ exp ⟩

Let o range over ⟨ inop ⟩ .

$[\![e_0\ o\ e_1]\!]$ = app $[\![o]\!]$ tup-seq($[\![e_0]\!]$,$[\![e_1]\!]$)

► *Complete expressions*

Complete expressions are all of the above and the ML constructions for conditionals.

⟨ exp ⟩ → ⟨ exp ⟩ 'andalso' ⟨ exp ⟩ | ⟨ exp ⟩ 'orelse' ⟨ exp ⟩
  | 'if' ⟨ exp ⟩ 'then' ⟨ exp ⟩ 'else' ⟨ exp ⟩

$[\![e_0$ andalso $e_1]\!]$ = cond $[\![e_0]\!]$ $[\![e_1]\!]$ ff
$[\![e_0$ orelse $e_1]\!]$ = cond $[\![e_0]\!]$ tt $[\![e_1]\!]$
$[\![$if e_0 then e_1 else $e_2]\!]$ = cond $[\![e_0]\!]$ $[\![e_1]\!]$ $[\![e_2]\!]$

B.2 Declarations

For declarations, let us introduce the relevant MSDF module that contains all CMSOS constructions related to declarations in ML.

```
msos Lang/ML/Dec is
 see Lang/ML/Exp' .

 see Cons/Prog, Cons/Prog/Dec .

 see Cons/Dec, Cons/Dec/bind, Cons/Dec/simult-seq,
     Cons/Dec/accum, Cons/Dec/local .

 see Cons/Exp, Cons/Exp/local .
sosm
```

► *let-Expressions*

We begin by extending the atomic expressions with the 'let-in-end' expression, which is mapped into the CMSOS 'local'.

$\langle\,\textsf{exp}\,\rangle \rightarrow$ ‘let’ $\langle\,\textsf{dec}\,\rangle$ ‘in’ $\langle\,\textsf{exp}\,\rangle$ ‘end’

Let d range over $\langle\,\textsf{dec}\,\rangle$.

$[\![\texttt{let d in } e \texttt{ end}]\!] = \texttt{local } [\![\mathsf{d}]\!]\ [\![e]\!]$

▶ *Value bindings*

Next, the declarations are defined. Currently, ML supports only value bindings, that are converted into the CMSOS ‘bind’ construction.

$\langle\,\textsf{dec}\,\rangle \rightarrow$ ‘val’ $\langle\,\textsf{vid}\,\rangle$ ‘=’ $\langle\,\textsf{exp}\,\rangle$

$[\![\texttt{val i = } e]\!] = \texttt{bind } [\![\mathsf{i}]\!]\ [\![e]\!]$

B.3 Imperatives

ML does not have the concept of a “command,” as everything is an expression, but it does have imperative features. Module ‘Lang/ML/Cmd’ defines this.

```
msos Lang/ML/Cmd is
 see Lang/ML/Dec .

 see Cons/Cmd, Cons/Cmd/seq-n, Cons/Cmd/effect, Cons/Cmd/while .

 see Cons/Exp, Cons/Exp/seq-Cmd-Exp, Cons/Exp/seq-Exp-Cmd,
     Cons/Exp/assign-seq, Cons/Exp/ref, Cons/Exp/assigned .

 see Cons/Var, Cons/Var/alloc, Cons/Var/deref .

 Storable ::= Value .
sosm
```

Next we add another “external” definition, which is the equation that allocates a new cell on a given store.

```
mod Lang/ML/Cmd' is
 including Lang/ML/Cmd .

 var Store : Store .
 eq new-cell (Store) = cell (length (Store) + 1) .
endm
```

▶ *Sequencing of expressions*

We begin by revisiting atomic expressions, where we define the syntax of sequencing of expressions. Sequences of expressions are converted into the '`seq-Cmd-Exp`' construction, which receives a sequence of commands and a final expression. Each command is an expression enclosed by the '`effect`' construction. For example, the sequence of expressions '`3;4;1`' is converted into '`seq seq (effect (3), effect (4)), 1`'.

⟨ exp ⟩ → '(' ⟨ exp seq ⟩ ';' ⟨ exp ⟩ ')'
⟨ exp seq ⟩ → ⟨ exp ⟩ ';' ⟨ exp seq ⟩

Let es range over ⟨ exp seq ⟩.

⟦(es ; e)⟧ = `seq (seq` ⟦es⟧`)` ⟦e⟧
⟦e ; es⟧ = `(effect` ⟦e⟧`,`⟦es⟧`)`

▶ *Dereferencing of expressions*

The dereferencing of expressions is straightforward: we first dereference the expression into a cell with the construction '`deref`' and return the assigned value to this cell with the construction '`assigned`'.

⟨ exp ⟩ → '`!`' ⟨ exp ⟩

⟦! e⟧ = `assigned (deref (`⟦e⟧`))`

▶ *Referencing of expressions*

The application expressions have the additional rule of the referencing of expressions. It first allocates a new cell on the store and encloses this cell with the '`ref`' construct so that it becomes a value.

⟨ exp ⟩ → '`ref`' ⟨ exp ⟩

⟦`ref` e⟧ = `ref (alloc (`⟦e⟧`))`

▶ *Assignment*

The infix expressions now have the assignment operation. Since an assignment in ML does not have any final value, we use the construction '`seq-Cmd-Exp`' to first execute the assignment and then to return the empty tuple ('`tup()`'). The assignment itself is made using the '`assign-seq`' construction by first dereferencing the assigned expression.

$\langle\,\textsf{exp}\,\rangle \rightarrow \langle\,\textsf{exp}\,\rangle$ ‘`:=`’ $\langle\,\textsf{exp}\,\rangle$

$[\![e_0 \texttt{ := } e_1]\!] = \texttt{seq (effect (assign-seq (deref } [\![e_0]\!]\texttt{) } [\![e_1]\!]\texttt{)) tup()}$

► *Loops*

Finally, we add a construction that is typical of imperative languages, which is the looping command.

$\langle\,\textsf{exp}\,\rangle \rightarrow \langle\,\textsf{while}\,\rangle \langle\,\textsf{exp}\,\rangle$ ‘`do`’ $\langle\,\textsf{exp}\,\rangle$

$[\![\texttt{while } e_0 \texttt{ do } e_1]\!] = \texttt{seq (while } [\![e_0]\!] \texttt{ (effect } [\![e_1]\!]\texttt{)) tup()}$

B.4 Abstractions

Let us introduce the relevant MSDF module, ‘`Lang/ML/Abs`’, to introduce the equations for abstractions.

```
msos Lang/ML/Abs is
 see Lang/ML/Dec .

 see Cons/Exp, Cons/Exp/Abs, Cons/Exp/close, Cons/Exp/app-seq .
 see Cons/Abs, Cons/Abs/abs-Exp, Cons/Abs/closure .
 see Cons/Par, Cons/Par/bind, Cons/Par/tup .
 see Cons/Dec, Cons/Dec/app, Cons/Dec/rec .
sosm
```

► *Recursive functions*

Our language only has recursive functions. The version here is a very simple for of recursive functions in ML that does not make use of its full pattern matching capabilities (we opted to show an example of anonymous functions—or lambda functions—and pattern matching rules in the semantics of the Mini-Freja language, Section 6.2). The new option for the ‘`dec`’ nonterminal shows the syntax of recursive functions: the first ‘`vid`’ is the name of the function, the second is the single argument, and the ‘`exp`’ is the body. It is converted into the binding of the function name to a closure.

$\langle\,\textsf{dec}\,\rangle \rightarrow$ ‘`fun`’ $\langle\,\textsf{vid}\,\rangle \langle\,\textsf{vid}\,\rangle$ ‘`=`’ $\langle\,\textsf{exp}\,\rangle$

$[\![\texttt{fun } \mathtt{i}_0\ \mathtt{i}_1 \texttt{ = } e]\!] = \texttt{rec (bind } [\![\mathtt{i}_0]\!] \texttt{ (close (abs (bind } [\![\mathtt{i}_1]\!]\texttt{) } [\![e]\!]\texttt{)))}$

B.5 Concurrency

Finally, let us present the concurrency primitives of ML. The following module, '`Lang/ML/Conc`', gathers the necessary MSDF modules.

```
msos Lang/ML/Conc is
 see Lang/ML/Cmd', Lang/ML/Abs .

 see Cons/Cmd, Cons/Cmd/send-chan-seq,
     Cons/Cmd/start .

 see Cons/Exp, Cons/Exp/recv-chan,
     Cons/Exp/alloc-chan .

 see Cons/Sys, Cons/Sys/Cmd, Cons/Sys/conc,
     Cons/Sys/conc-chan, Cons/Sys/quiet .
sosm
```

▶ *Creating new threads*

The operation '`spawn`' creates a new thread of execution, and is equivalent to the '`start`' construction from CMSOS.

⟨ exp ⟩ → '`spawn`' ⟨ exp ⟩

⟦`spawn` e⟧ = `seq (start` ⟦e⟧`)`

▶ *Creating new channels*

The declaration '`chan`' creates a new channel and binds to the identifier passed as its argument.

⟨ dec ⟩ → '`chan`' ⟨ vid ⟩

⟦`chan i`⟧ = `bind` ⟦`i`⟧ `alloc-chan`

▶ *Sending and receiving through channels*

The operations '`send`' and '`receive`' transmit information over a channel and are implemented, respectively, by the CMSOS constructions '`send-chan-seq`' and '`recv-chan`'.

⟨ exp ⟩ → '`send`' '`(`' ⟨ exp ⟩ '`,`' ⟨ exp ⟩ '`)`' | '`receive`' ⟨ exp ⟩

$[\![$`send ( `e_0` , `e_1` )`$]\!]$ = `seq (send-chan-seq `$[\![e_0]\!]$ $[\![e_1]\!]$`) tup()`
$[\![$`receive `$e$$]\!]$ = `recv-chan `$[\![e]\!]$

► *Complete concurrent ML programs*

Now, the final rule. All ML programs that are concurrent must be prefixed by '`cml`'. It is converted to the '`quiet`' CMSOS construction.

⟨ exp ⟩ → '`cml`' ⟨ exp ⟩

$[\![$`cml `$e$$]\!]$ = `quiet (effect `$[\![e]\!]$`)`

APPENDIX C – MiniJava specification

This Chapter contains the complete MiniJava specification that was described in Section 6.1.3.

C.1 Expressions

Expressions consist of mathematical operations, identifiers, method invocations (that always return a value), literals, and objects themselves.

⟨ exp ⟩ → ⟨ math operation ⟩ | ⟨ id ⟩ | ⟨ method invocation ⟩
  | ⟨ literal ⟩ | ⟨ this ⟩ | ⟨ new ⟩

▶ *Math operations*

⟨ math operation ⟩ → ⟨ exp ⟩ ⟨ math op ⟩ ⟨ exp ⟩
⟨ math op ⟩ → ‘&&’ | ‘<’ | ‘+’ | ‘/’ | ‘%’ | ‘–’ | ‘*’ | ‘>’ | ‘<=’ | ‘>=’ | ‘=’

Let e_i range over ⟨ exp ⟩, and m range over ⟨ math op ⟩.

$$[\![e_0 \;\; m \;\; e_1]\!] \;=\; \texttt{app}\ [\![m]\!]\ \texttt{tup-seq}\ ([\![e_0]\!],\ [\![e_1]\!])$$

▶ *Identifiers*

Let i range over ⟨ id ⟩. When i is not the left-hand side of an assignment:

$$[\![i]\!] \;=\; \texttt{assigned (deref}\ i\texttt{)}$$

Otherwise, it is as follows:

$$[\![i]\!] \;=\; \texttt{deref}\ i$$

▶ *Method invocations*

⟨ method invocation ⟩ → ⟨ exp ⟩ '.' ⟨ id ⟩ '(' ⟨ exp ⟩*')'

$[\![e\ .\ i\ (\ e*\ )]\!] = \texttt{app (app nth(}n\texttt{)}\ [\![e]\!]\texttt{) (tup-seq p)}$

The evaluation of e must return an object; n is the method number in the class of the object returned by e, obtained by looking up the method name i in the metaclass information generated in the static analysis phase of the compilation process; and p is constructed as a sequence of `ref (alloc` $[\![e_i]\!]$`)` which allocates a new memory entry for each parameter e_i in e∗.

▶ *Literals*

⟨ literal ⟩ → ⟨ boolean literal ⟩ | ⟨ integer literal ⟩
⟨ boolean literal ⟩ → 'true' | 'false'

Literals are converted verbatim to CMSOS constructions.

▶ *Self-reference*

⟨ this ⟩ → 'this'

$[\![\texttt{this}]\!] = \texttt{app self tup()}$

▶ *Object instantiations.*

⟨ new ⟩ → 'new' ⟨ id ⟩ '()'

$[\![\texttt{new i ()}]\!] = \texttt{app i tup()}$

C.2 Statements

MiniJava contains the usual statements of imperative programming languages: conditionals, loops, output, assignment, etc.

⟨ statement ⟩ → ⟨ if ⟩ | ⟨ while ⟩ | ⟨ block ⟩ | ⟨ print ⟩ | ⟨ assign ⟩ | ⟨ empty ⟩

▶ *Conditionals*

⟨ if ⟩ → ‘`if`’ ⟨ exp ⟩ ‘`then`’ ⟨ statement ⟩ ‘`else`’ ⟨ statement ⟩

Let s_i range over ⟨ statement ⟩.

$$[\![\texttt{if}\ e\ \texttt{then}\ s_0\ \texttt{else}\ s_1]\!] = \texttt{cond}\ [\![e]\!]\ [\![s_0]\!]\ [\![s_1]\!]$$

▶ *Loops*

⟨ while ⟩ → ‘`while`’ ‘(’ ⟨ exp ⟩ ‘)’ ⟨ statement ⟩

$$[\![\texttt{while (} e \texttt{) } s]\!] = \texttt{while}\ [\![e]\!]\ [\![s]\!]$$

▶ *Block statements*

⟨ block ⟩ → ‘{’ (⟨ statement ⟩)*‘}’

$$[\![\{\ s*\ \}]\!] = \texttt{seq}\ [\![s*]\!]$$

▶ *Output*

⟨ print ⟩ → ‘`System.out.println`’ ‘(’ ⟨ exp ⟩ ‘)’

$$[\![\texttt{System.out.println (} e \texttt{)}]\!] = \texttt{print}\ [\![e]\!]$$

▶ *Assignment*

⟨ assign ⟩ → ⟨ exp ⟩ ‘=’ ⟨ exp ⟩

$$[\![e_0\ \texttt{=}\ e_1]\!] = \texttt{effect (assign-seq}\ [\![e_0]\!]\ [\![e_1]\!]\texttt{)}$$

▶ *Empty statement*

⟨ empty ⟩ → ‘;’

$$[\![;]\!] = \texttt{skip}$$

C.3 Classes

▶ *Class declaration*

As described at the beginning of this Section a class declarations defines a "prototype" object, which is a closure where the fields become bindings and the methods become projections of a tuple.

⟨ class declaration ⟩ → 'class' ⟨ identifier ⟩ '{' (⟨ field declaration ⟩)*
(⟨ method declaration ⟩)*'}'

Let f_i range over ⟨ field declaration ⟩ and m_i over ⟨ method declaration ⟩.

```
[[class i { f* m* }]] =
   local
    (accum (accum [[f*]])
           (rec (bind self
                 close (abs (bind dummy) tup-seq ([[m]])))))
    app self tup()
```

▶ *Main class declaration*

⟨ main class declaration ⟩ → 'class' ⟨ identifier ⟩
'{' ⟨ main method declaration ⟩ '}'

```
[[class i { m }]] =
   local
    (accum void
           (rec (bind self
                 close (abs (bind dummy) tup-seq ([[m]])))))
    app self tup()
```

The main difference is the lack of field declarations and the existence of a single method.

▶ *Field declarations*

The type information is used only on the static analysis phase. Since the only primitive type is the integer, we bind the identifier i to a newly allocated cell, with the default value of zero.

⟨ field declaration ⟩ → ⟨ type ⟩ ⟨ identifier ⟩

Let t range over ⟨ type ⟩.

```
⟦t i⟧  = bind i (ref (alloc 0))
```

▶ *Method declaration*

A method declaration is converted into a closure with parameters being bound simultaneously by the ‘tup’ construction. The closure body is a ‘local’ definition with the variable declarations as the declaration part the method body as the expression being evaluated. We use the ‘seq-Cmd-Exp’ command so that the last expression evaluated is returned as the method return value.

⟨ method declaration ⟩ → ⟨ type ⟩ ⟨ identifier ⟩ ‘(’ ((⟨ parameter ⟩)*‘)’
‘{’ ((⟨ var declaration ⟩)*((⟨ statement ⟩)*‘return’ ⟨ expression ⟩ ‘}’

Let p_i range over ⟨ parameter ⟩.

```
⟦t i ( p∗ ) { v∗ s∗ return e }⟧ =
 close (abs tup(⟦p∗⟧) (local (accum ⟦v∗⟧) (seq (seq ⟦s∗⟧) e)))
```

▶ *Main method declaration*

⟨ main method declaration ⟩ → ‘public static void main (String arg[]) {’
((⟨ statement ⟩)*‘}’

```
⟦public static void main (String arg[]) { s∗ }⟧ =
   close (abs tup(@) (local void (seq (seq ⟦s∗⟧) 0)))
```

The main method is similar to the generic method declaration, except that it does not take arguments, variables declarations, nor has any return expression. In this case the return expression is assumed to be zero.

▶ *Complete program*

Now we must close this specification by creating the main body of the program. It consists of a series of bindings from the class names to the prototype objects obtained from the class declarations. The body is a call to the zeroth projection of the main class, which is exactly the main method.

Let cd_i be a class, not the main, declaration and *exec* the body of the main program.

```
⟦goal⟧  = local (accum ⟦cd∗⟧) ⟦exec⟧
```

The conversion of the class declarations is a series of bindings from class names (represented by n_i) to the prototype objects (represented by o_i). Thus, for some class declaration i, $[cd_i]$ is as follows:

$$[\![cd_i]\!] \ = \texttt{bind}\ n\ o_i$$

Now, the equation for $exec$. It is a call to the main method of the main class. Even though this method does not receive any arguments, an empty tuple with a single reference is passed as a "dummy" parameter. The prototype object of the main class is represented by o_m.

```
⟦exec⟧ =
 (app (app nth(0) o_m)
      (tup-seq (ref (alloc 0))))
```

To complete the semantics and show an example of execution we show next the MSDF module that gathers all relevant CMSOS constructions necessary for the MiniJava language:

```
msos MiniJava is
 see Cons/Prog, Cons/Prog/Exp .
 see Cons/Exp, Cons/Exp/Boolean, Cons/Exp/Int, Cons/Exp/local,
     Cons/Exp/Id, Cons/Exp/cond, Cons/Exp/app-Op,
     Cons/Exp/app-Id, Cons/Exp/tup, Cons/Exp/tup-seq .
 see Cons/Arg, Cons/Arg/Exp .
 see Cons/Op .
 see Cons/Id .
 see Cons/Prog, Cons/Prog/Dec .
 see Cons/Dec, Cons/Dec/bind, Cons/Dec/simult-seq,
     Cons/Dec/accum, Cons/Dec/local .
 see Cons/Exp, Cons/Exp/local .
 see Cons/Cmd, Cons/Cmd/seq-n, Cons/Cmd/effect, Cons/Cmd/while,
     Cons/Cmd/print .
 see Cons/Exp, Cons/Exp/seq-Cmd-Exp, Cons/Exp/seq-Exp-Cmd,
     Cons/Exp/assign-seq, Cons/Exp/ref, Cons/Exp/assigned .
 see Cons/Var, Cons/Var/alloc, Cons/Var/deref .
 see Cons/Exp/ref .
 see Cons/Exp, Cons/Exp/Abs, Cons/Exp/close, Cons/Exp/app-seq .
 see Cons/Abs, Cons/Abs/abs-Exp, Cons/Abs/closure .
 see Cons/Par, Cons/Par/bind, Cons/Par/tup .
 see Cons/Dec, Cons/Dec/app, Cons/Dec/rec .

 Bindable ::= Value | Op .
 Op ::= nth (Int) | plus | times | minus | eq | lt | gt .
 Passable ::= Value .
 Storable ::= Value .
sosm
```

Next, the following code defines the initial basis for MiniJava programs, with the initial record and the '`apply-op`' and '`new-cell`' equations. We also define an '`output`' function which receives a configuration as argument and that, once the computation end with a '`skip`' command, it removes the output that appears on the write-only component '`out`'.

```
mod MiniJava' is
 including MiniJava .

 var I : Int .
 var V : Value .
 var VL : Seq`(Value`) .

 op init-rec : -> Record .
 eq init-rec = { out' = (()).Seq`(Value`),
                env = (void).Map`(Id`|`Bindable`),
                store = (void).Map`(Cell`|`Storable`) } .

 vars i1 i2 : Int .
 eq apply-op (plus, (i1, i2)) = i1 + i2 .
 eq apply-op (minus, (i1, i2)) = i1 - i2 .
 eq apply-op (times, (i1, i2)) = i1 * i2 .
 eq apply-op (eq, (i1, i2)) = if i1 == i2 then tt else ff fi .
 eq apply-op (lt, (i1, i2)) = if i1 < i2 then tt else ff fi .
 eq apply-op (gt, (i1, i2)) = if i1 > i2 then tt else ff fi .
 eq apply-op (nth (0),  (V, VL)) = V .
 ceq apply-op (nth (I),  (V, VL))
 = apply-op (nth (I - 1),  (VL))
 if I > 0 .

 op ide : Qid -> Id .
 op ide : Op -> Id .

 eq init-env = (ide(eq) |-> eq +++ ide(lt) |-> lt +++
               ide(gt) |-> gt +++ ide(plus)  |-> plus +++
               ide(times) |-> times +++ ide(minus) |-> minus) .

 var Store : Store .
 eq new-cell (Store) = cell (length (Store) + 1) .

 sort Output .
 op output : Conf -> Output .
 op output : Seq`(Value`) -> Output .

 rl output(< V:Value ::: 'Exp, { out' = VL:Seq`(Value`),
    PR:PreRecord } >) => output(VL:Seq`(Value`)) .

endm)
```

APPENDIX D – Mini-Freja specification

This Chapter contains the complement to the Mini-Freja semantics that was described in Section 6.2, with addition of the rules for pattern matching.

Mini-Freja has pattern matching capabilities similar to those of Standard ML. The pattern matcher is the '`case Exp of Rules`' construction, where '`Rules`' is a sequence of options to be matched, each with a resulting expression.

```
Rules .
Rule .

Rule ::= Pat => Exp .

Rules ::= Rule
        | Rules || Rules [assoc] .
```

Each rule is a pattern to be matched against, and the resulting expression. Patterns follow the same syntax of expressions: we may match against variables, constants, and lists.

```
Pat .
Pat ::= Pat :: Pat [assoc] .
Pat ::= p Const | p Var .
```

Unfortunately, due to preregularity issues, we need the coercion function '`p_`' that lifts constants and variables into the set of patterns. The problem happens with the list operator '`Pat :: Pat`': had we not used the coercion function '`p_`', a term like '`3 :: 5`' would not have a *least sort*, since it could be either a '`Pat`' or a '`Exp`'.

The following rules implement the pattern matcher of the Mini-Freja language. We begin by creating an additional operation '`case(v,R)`', that matches a value `v` (obtained from an expression) against a set `R` of rules.

```
 Value ::= case (Value, Rules) .

        Exp ==> Value,
              case (Value, Rules) ==> Value'
 [case] -- ---------------------------------
        case Exp of Rules : Exp ==> Value' .
```

Matches are defined according to the following signature. When a match is successful, it evaluates to '`myes(ρ)`', where ρ is the binding resulting from the match. Otherwise, it evaluates to '`mno`'.

```
Match .
Match ::= myes (Env) | mno .
```

First, the base case, where the set R of rules consists of a single rule. The function '`match(v,p)`' matches v against pattern p and returns either '`myes(ρ)`', if the match is successful, or '`mno`' otherwise. If the value '`Value`' matches the pattern '`Pat`', then the expression '`Exp`' is evaluated by overriding the current environment with ρ.

```
Match ::= match (Value, Pat) .

        match (Value, Pat) ={env = Env, -}=> myes(Env'),
        Env'' := Env' / Env, Exp ={env = Env'', -}=> Value'
[case1] -- ---------------------------------------------
        case (Value, Pat => Exp) : Value
                                  ={env = Env, -}=> Value' .
```

Now let us see the general case, in which there is at least two rules in R. The rule below specifies the following: the result of the matching of '`Value`' against '`Pat`' is given on to the '`case-choose`' function, which will return a value '`Value'`'

```
        match (Value, Pat) ==> Match,
        case-choose (Match, Exp, Value, Rules) ==> Value'
[case2] -- ------------------------------------------
        case (Value, ((Pat => Exp) || Rules)) : Value
                                               ==> Value' .
```

Let's see how the auxiliary function '`case-choose(m,e,v,R)`' works. If the match m is '`mno`', it means that the matching against the first rule of R was unsuccessful, so the value must be matched against the remainder of the rules.

```
Value ::= case-choose (Match, Exp, Value, Rules) .

              case (Value, Rules) ==> Value'
[case-choose] -- ----------------------------------------
              case-choose (mno, Exp, Value, Rules) : Value
                                                 ==> Value' .
```

Otherwise, if the matching is successful, '`case-choose`' works in a similar way to the rule '`[case1]`'. The environment '`Env'`' obtained from the successful match overrides the current environment to evaluate '`Exp`' to a final value '`Value''`'.

```
                Env'' := Env' / Env,
                Exp ={env = Env'', -}=> Value'
[case-choose] -- -------------------------------------------
                case-choose (myes(Env'),Exp,Value,Rules) : Value
                                     ={env = Env, -}=> Value' .
```

The following rules specify each possible case of matching of values against patterns. First, matching a value against a pattern that is a variable is always successful and creates a binding from the variable to the value.

```
[match-var] match (Value, p Var) : Match
            ={env = Env, -}=> myes (Var |-> Value / Env) .
```

Matching a constant against a constant is successful only if both constants are equal.

```
                    Match := if Const1 == Const2
                             then myes (Env)
                             else mno fi
[match-const-const] -- ------------------------------
                    match (Const1, p Const2) : Match
                             ={env = Env, -}=> Match .
```

Matching a '`cons`' against a list makes needs an auxiliary function '`match-pair(m,v,p)`' that will iterate through the list, gathering the bindings, if the matches are successful.

```
Match ::= match-pair (Match, Value, Pat)
```

Rule '`[match-cons-cons]`' states that, when matching a list against another we first attempt to match the elements at the beginning of both lists, and then use '`match-pair`' to, in a big-step manner, match the remainder of both lists.

```
                  match (Value1, Pat1) ==> Match1,
                  match-pair (Match1, Value2, Pat2) ==> Match2
[match-cons-cons] -- ----------------------------------------
                  match (cons (Value1, Value2),
                         Pat1 :: Pat2) : Match ==> Match2 .
```

Notice that, on rule '`[match-cons-cons]`', '`match-pair`' receives the matching of the first elements of both lists ('`Match1`'). If this matching is unsuccessful, then the entire list matching fails also.

```
[match-pair] match-pair (mno, Value, Pat) : Match ==> mno .
```

Otherwise it recursively matches against the remainder of the list, gathering the bindings during the process

```
             Env'' := Env' / Env,
             match (Value, Pat) ={env = Env'', -}=> Match
[match-pair] -- ---------------------------------------------
             match-pair (myes (Env'), Value, Pat) : Match
                                    ={env = Env, -}=> Match .
```

Matching a constant against a list or a list against a constant always fails.

```
[match-const-cons] match (Const, Pat1 :: Pat2) : Match
                   ==> mno .
[match-cons-const] match (cons(Value1,Value2),p Const) : Match
                   ==> mno .
```

APPENDIX E – Distributed algorithms

This Appendix completes Chapter 6.3 with more examples of distributed algorithms. Section E.1 describes a specification of a mutual exclusion algorithm using semaphores; Section E.2 continues the specification of the Dining Philosophers on Section 6.3.2.2 with some additional specifications and verifications; Section E.3 shows Lamport's Bakery Algorithm for mutual exclusion, verified using an equational abstraction [46, 47]; finally, Section E.4 shows more examples of model checking by specifying an algorithm for leader election on an asynchronous ring.

E.1 Mutual exclusion using semaphores

This Section specifies a mutual exclusion algorithm using semaphores. It is also an introductory example that shows how a process keeps is internal state using the 'St' set.

Let us begin with a specification without semaphores and check the race condition problem, in this specification processes have two possible states: either they are inside the critical region ('cric') or not, the remainder region ('rem').

```
 St .
 St ::= crit | rem .

 Proc .
 Proc ::= pid (Int, St) .
```

Processes keep entering and leaving their critical region.

```
prc (Int, rem) : Proc --> prc (Int, crit) .
prc (Int, crit) : Proc --> prc (Int, rem) .
```

This is specification is simple enough and we may search for all possible states. A simple 'search' command suffices to show all four options:

```
(search
  (< (prc (0, rem) prc (1, rem)) ::: 'Soup,
     { null } >) =>* C:Conf .)
```

```
Solution 1
C:Conf <- <(prc(0,rem) prc(1,rem))::: 'Soup,{null}>

Solution 2
C:Conf <- <(prc(0,crit) prc(1,rem))::: 'Soup,{null}>

Solution 3
C:Conf <- <(prc(0,rem) prc(1,crit))::: 'Soup,{null}>

Solution 4
C:Conf <- <(prc(0,crit) prc(1,crit))::: 'Soup,{null}>
```

To avoid the race condition shown on the fourth solution, let us rewrite our rules with a semaphore semantics: before entering the critical region, a process will go through intermediate states '`down`' and '`up`', represented by the read-write component '`sem`'.

```
Label = { sem : Int, sem' : Int, ... } .
```

We need to at '`down`' and '`up`' to our set of possible states, '`St`':

```
 St ::= down | up .
```

Before entering a critical region, a process first go to its '`down`' state:

```
 prc (Int, rem) : Proc --> prc (Int, down) .
```

Following the semaphore semantics, a process will only access its critical region when the semaphore is zero.

```
    Int' == 0
 -- -------------------------------------------------------
 prc (Int, down) : Proc -{sem = Int', sem' = Int', -}->
 prc (Int, down) .

    Int' > 0, Int'' := Int' - 1
 -- ----------------------------------------------------------
 prc (Int, down) : Proc -{sem = Int', sem' = Int'', -}->
 prc (Int, crit) .
```

Moving from the critical region to the remainder, the process first executes its '`up`' action, incrementing the value of the semaphore by one.

```
prc (Int, crit) : Proc --> prc (Int, up) .

   Int'' := Int' + 1
-- ------------------------------------------------------
prc (Int, up) : Proc -{sem = Int', sem' = Int'', -}->
prc (Int, rem) .
```

Now, a search for a configuration where a race condition occurs is unsuccessful.

```
search : <(prc(0,rem)prc(1,rem))::: 'Soup,{sem = 1}> =>*
         <(prc(0,crit)prc(1,crit))::: 'Soup,R:Record > .

No solution.
```

Let us use the model checker to confirm this result. We begin by creating an auxiliar operation '`create-conf(i)`' that creates a configuration with `i` processes. The proposition '`race-condition`' holds whenever is more than one process is its critical zone.

```
rewrites: 817190 in 7638ms cpu (7638ms real)
          (106978 rewrites/second)
reduce in CHECK :
  modelCheck(create-conf(10), [] ~ race-condition)
result Bool :
  true
```

It is interesting to observe that, since the system does not have justice, there is a possibility that a process may *never* enter its critical region. Let us add a new proposition '`in-crit(i)`' that holds when a process `i` is in its '`crit`' state. The following verification fails with a counterexample where process '`1`' is "stuck" on its '`down`' state.

```
reduce in CHECK :
  modelCheck(create-conf(3), <> in-crit(1))
result ModelCheckResult :
  counterexample

{prc (1, rem) prc (2, rem) prc (3, rem)}
{prc (1, down) prc (2, rem) prc (3, rem)}
{prc (1, down) prc (2, down) prc (3, rem)}
{prc (1, down) prc (2, crit) prc (3, rem)}
{prc (1, down) prc (2, up)  prc (3, rem)}
{prc (1, down) prc (2, up)  prc (3, down)}
{prc (1, down) prc (2, rem) prc (3, down)}
{prc (1, down) prc (2, down) prc (3, down)},
{prc (1, down) prc (2, crit) prc (3, down)}
```

E.2 Dining Philosophers

This Section completes Section 6.3.2.2 with the following additional material: the remainder for the rules for the specification of Dining Philosophers; a variant of the specification in which the philosophers only eat once; another variant of the algorithm with a fair scheduling; and an incorrect specification which is analyzed and verified to be incorrect using Maude's formal tools.

E.2.1 Remainder of the rules

This Section shows the remainder of the rules not shown in Section 6.3.2.2, that is, the rules for the states '`stest-left`', '`sleave-try`', '`scrit`', and '`srem`'.

The rule for the '`stest-left`' state is similar to the rule for '`stest-right`'. The difference is that, when the left fork is acquired, the process moves to '`sleave-sty`'.

```
odd (Int),
Pids := lookup (((Int + 1) rem n), Queue),
Pids' := if (not Int in Pids)
         then insert-back (Int, Pids)
         else Pids fi,
Queue' := ((((Int + 1) rem n)) |-> Pids') / Queue,
                                     first (Pids') == Int
-- ------------------------------------------------------
prc (Int, stest-left) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, sleave-try) .
```

One in the '`sleave-sty`' state, a process moves to is critical region.

```
                    odd (Int)
-- ----------------------------------------------
prc (Int, sleave-try) : Proc --> prc (Int, scrit) .
```

After accessing its critical region, a process moves to the '`sexit`' state, in which it first puts the right fork down, and then the left.

```
                     odd (Int)
-- ----------------------------------------------
prc (Int, scrit) : Proc -{-}-> prc (Int, sexit) .

   odd (Int)
-- ----------------------------------------------
prc (Int, sexit) : Proc --> prc (Int, sreset-right) .
```

In order to put the right fork down, it must remove itself from the queue on that fork. Since the queue only had the pid of the process, it will be empty after this operation.

```
odd (Int), Pids := lookup (Int, Queue),
Pids' := remove (Int, Pids),
Queue' := (Int |-> Pids') / Queue
-- ----------------------------------
prc (Int, sreset-right) : Proc
   -{q = Queue, q' = Queue', -}->
              prc (Int, sreset-left) .
```

The same process is make for the left fork.

```
odd (Int), Pids := lookup (((Int + 1) rem n), Queue),
Pids' := remove (Int, Pids),
Queue' := ((((Int + 1) rem n)) |-> Pids') / Queue
-- ------------------------------------------------------
prc (Int, sreset-left) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, sleave-exit) .
```

One the left fork is taken down, a process goes to its '`srem`' state, which models the philosopher thinking.

```
                    odd (Int)
-- ------------------------------------------------
prc (Int, sleave-exit) : Proc --> prc (Int, srem) .
```

After thinking for a while a philosopher gets hungry again and returns to its '`stry`' state.

```
                  odd (Int)
-- ------------------------------------------
prc (Int, srem) : Proc --> prc (Int, stry) .
```

E.2.2 Dining Philosophers, terminating specification

This Section presents a variant of the specification in which each philosopher, after eating, prints out its pid and stops. This specification was inspired by the one present on [20].

The specification is similar to the one shown in Sections 6.3.2.2 and E.2.1 with some modifications. The first is the addition of a write-only component '`Int*`', indexed by '`out'`', to model the output of information by the processes.

```
Label = {out' : Int*, q : Queue, q' : Queue, ...} .
```

We also change the rule for the '`scrit`' state, making the process output its pid.

```
                    odd (Int)
 -- ----------------------------------------------------------
 prc (Int, scrit) : Proc -{out' = Int, -}-> prc (Int, sexit) .
```

The following rule for the 'srem' state is removed, since, a philosopher no longer gets hungry again after thinking.

```
            odd (Int)
 -- ----------------------------------------
 prc (Int, srem) : Proc --> prc (Int, stry) .
```

This slight modification of the algorithm allows for more interesting verifications. Searching for all final states using the 'search' command, we must arrive in states in which the 'out'' component contains all the pids of the processes in the configuration.

```
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <- <( prc(0,srem) prc(1,srem) prc(2,srem) prc(3,srem))
 {..., out' = 0,1,2,3}>

Solution 2
C:Conf <- <( prc(0,srem) prc(1,srem) prc(2,srem) prc(3,srem))
 {...,out' = 0,1,3,2}>

Solution 3
C:Conf <- <( prc(0,srem) prc(1,srem) prc(2,srem) prc(3,srem))
 {...,out' = 0,3,1,2}>
...
```

There are several possible variations of the contents of the 'out'' component, since the order in which a philosopher eats is non-deterministic.

Following the example in [20] let us model check this specification using a proposition 'check(i)' which holds when the component 'out'' contains all numbers less than i.

```
rewrites: 505248 in 2450ms cpu (2440ms real)
          (206223 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> check (n - 1))
result Bool :
  true
```

In this case, since a process eventually stops, all processes eventually eat. The example below shows the case of process '0'. Recall that 'state(i,s)' holds then process i is in state s.

```
rewrites: 176389 in 1750ms cpu (1750ms real)
          (100793 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state (0,scrit))
result Bool :
  true
```

E.2.3 Fair scheduling

It is interesting to see what is the effect of adding a fair scheduling, according to the discussion on Section 6.3.1.2, on the specification. Let us make these changes to the terminating specification (Section E.2.2), but they are easily adapted to the looping specification. Besides the scheduling rules, of course, the only change to that specification is the addition of the following rule:

```
                   odd (Int)
 -- ----------------------------------------
 prc (Int, srem) : Proc --> prc (Int, srem) .
```

This is necessary because a process needs to pass its turn to the next process when it is in its stopped mode.

The interesting result of this change is that the verification capacity is greatly enhanced. For example, let us check a 200-philosopher configuration for a deadlock. Notice that there is no final state, now that, upon termination, process keep "passing the turn" indefinitely.

```
rewrites: 86864 in 10442ms cpu (10501ms real)
          (8318 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

No solution.
```

The model checking of the '`check(i)`' proposition is also successful.

```
rewrites: 1661019 in 42601ms cpu (43419ms real)
          (38989 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> check(n - 1))
result Bool :
  true
```

As it was expected with a fair scheduling of the execution, a process will now eventually eat.

```
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state(0,scrit))
result Bool :
  true
...
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state(199,scrit))
result Bool :
  true
```

E.2.4 An incorrect specification

This Section shows how a deadlock is detected in an incorrect specification, which, as we outlined on Section 6.3.2.2, is one that does not break the symmetry on the order on which each philosopher acquire its fork.

We may "break" any of the specifications described so far by removing the subset of the rules that applies to odd (or even) processes and, of course, removing the predicate '`odd(i)`' (or '`even(i)`') from the condition on the rules. Clearly, this should lead to a deadlock. In what follows we attempt to verify this deadlock with each of the several variants of the solution we developed.

Let us begin with the looping specification, in which each philosopher gets hungry again after thinking. A search for a final state with the '`search`' command with a configuration with four philosophers finds the deadlocked state: all philosophers are "stuck," holding their left forks.

```
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <-
   < prc (0,stest-left) prc (1,stest-left)
     prc (2,stest-left) prc (3,stest-left))
     { q = (0 |->[0] +++ 1 |->[1] +++
            2 |->[2] +++ 3 |->[3]) } >

No more solutions.
```

With the terminating specification, the search finds not only the states in which the philosophers successfully eat, but also the deadlock state ('`Solution 1`').

```
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <-
  < prc (0, stest-left) prc (1, stest-left)
    prc (2, stest-left) prc (3, stest-left)),
```

```
    { fair = 0,out’ = (),
      q = (0 |-> [0] +++ 1 |-> [1] +++
           2 |-> [2] +++ 3 |-> [3]) }>

Solution 2
C:Conf <-
  < prc (0, srem) prc (1, srem)
    prc (2, srem) prc (3, srem),
    { fair = 0, out’ = 0,1,2,3,
      q =(0 |-> [] +++ 1 |-> [] +++
          2 |-> [] +++ 3 |-> []) }>

Solution 3
C:Conf <-
  < prc (0, srem) prc (1, srem)
    prc (2, srem) prc (3, srem),
    { fair = 0, out’ = 0,1,3,2,
      q =(0 |-> [] +++ 1 |-> [] +++
          2 |-> [] +++ 3 |-> []) }>
...
```

The specification with the round-robin scheduling is also prone to the deadlock.

```
rewrites: 671 in 20ms cpu (20ms real) (33550 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <-
   < prc (0,stest-left) prc (1,stest-left)
     prc (2,stest-left) prc (3,stest-left)),
     { fair = 0, out’ = (),
       q =(0 |->[0]+++ 1 |->[1]+++ 2 |->[2]+++ 3 |->[3]) }>
```

Recall that, with a fair scheduling policy, the model checking of a proposition that states that eventually a process will enter its critical region succeeds. The following shows that this is no longer the case, and presents as counterexample the same deadlocked situation, omitted here for brevity.

```
rewrites: 2520 in 50ms cpu (50ms real) (50400 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state(0,scrit))
result ModelCheckResult :
  counterexample(...)
```

E.3 Bakery algorithm

This Section presents a specification of Lamport's Bakery Algorithm, described in [36]. Its primary objective it to give a verification example of an unbounded algorithm using an abstraction [46, 47]. Intuitively, the algorithm simulates a bakery (in Lamport's conception of how a bakery works) where customers wait for their turn by drawing tickets when they enter and are served in the order of their ticket numbers.

Let us begin the formal description by defining two read-write components: 'ch' models whether a process is choosing its number or not; 'nm' holds the chosen number. Both components are of the same type, 'IntM', which is a map from integers (the pids) to integers (the chosen numbers).

```
IntM = (Int, Int) Map .
Label = {ch : IntM, ch' : IntM,
         nm : IntM, nm' : IntM, ...} .
```

A process may go through the following states, explained throughout the transition rules.

```
St .
St ::= choosing (Int, Int)
     | waiting (Int)
     | rem
     | crit
     | try
     | exit .
```

When a process wants to go into its critical region, it tells others that it is doing so by changing its entry on the 'ch' component to '1'. The process then choose a number that is greater than all the number chosen by other processes. This is done in the 'choosing(i,m)' state, which i contains the number of processes left to check and m the greatest number found so far. Let us assume that the constant 'n' will be bound, by an equation, to the number of processes currently running.

```
           IntM' := (Int |-> 1) / IntM
-- ---------------------------------------------------
prc (Int, try) : Proc -{ch = IntM, ch' = IntM', -}->
               prc (Int, choosing (n - 1, -1)) .
(Int1 >= 0), (Int1 =/= Int), (Int2' := lookup (Int1, IntM)),

Int3 := if Int2' > Int2 then Int2' else Int2 fi
-- ----------------------------------------------------------
prc (Int, choosing (Int1, Int2)) : Proc
                      -{nm = IntM, nm' = IntM, -}->
                      prc (Int, choosing (Int1 - 1, Int3)) .
```

During the choosing process, a process must ignore its own number.

```
prc (Int, choosing (Int, Int2)) : Proc -->
               prc (Int, choosing (Int - 1, Int2)) .
```

When $i = -1$, the greatest number found is m. The process then chooses as its own number $m + 1$ and goes to the next phase of the algorithm.

```
Int'' := (Int' + 1), IntM'1 := (Int |-> 0) / IntM1,
IntM'2 := (Int |-> Int'') / IntM2
-- -------------------------------------------------
prc (Int, choosing (-1, Int')) : Proc
               -{ch = IntM1, ch' = IntM'1,
                 nm = IntM2, nm' = IntM'2, -}->
                           prc (Int, waiting (0)) .
```

On this phase, a process keeps a constant watch on the other processes, iterating through the '`waiting(i)`' state, where $0 \leq i \leq n - 1$ (recall that n is the number of processes). It waits until its number if the lowest of all in order to access its critical region and avoids comparing with any process that is currently choosing its own number.

Since there is a possibility that *several* processes begin the choosing process at the same time, it may happen that processes choose the same number. In order to deal with this, the comparison to find the lowest number is made lexicographically using (i, p) where i is the process number and p its pid. This is formalized by the transition below, which specifies that process '`Int`' is comparing its number with process '`Int'`'. The predicate '`Int1' == 0`' first makes sure that process '`Int'`' is not choosing a number. If the chosen number of process '`Int'`' is zero ('`Int2' == 0`'), process '`Int'`' just left the critical region and process '`Int`' may access it directly, otherwise the lexicographical comparison is made.

```
prc (Int, waiting (Int)) : Proc -->
    prc (Int, waiting ((Int + 1) rem n)) .

  Int' =/= Int,
  (Int1' := lookup (Int', IntM1)),
  (Int2' := lookup (Int', IntM2)),
  (Int2  := lookup (Int, IntM2)),
  St := if Int1' == 0 and
           (Int2' == 0 or
            ((Int2 < Int2') or
             (Int2 == Int2' and Int < Int')))
        then crit else waiting ((Int' + 1) rem n)
        fi
 -- ---------------------------------------------
 prc (Int, waiting (Int')) : Proc
               -{ch = IntM1, ch' = IntM1,
                  nm = IntM2, nm' = IntM2, -}->
                                 prc (Int, St) .
```

Upon exiting its critical region, a process, as we said, changes its chosen number to zero and moves to its '`rem`' state. Once in its '`rem`' state, a process attempts to access the critical region again by moving to its '`try`' state.

```
                 IntM2' := (Int |-> 0) / IntM2
 -- --------------------------------------------------------
 prc (Int, crit) : Proc -{nm = IntM2, nm' = IntM2', -}->
                                              prc (Int, rem) .

 prc (Int, rem) : Proc --> prc (Int, try) .
```

Unfortunately, this algorithm does not have an upper bound on the chosen number. Also, the apparently trivial solution of using integers modulo some very large `b` also fails.

We may verify that there is no upper bound on the chosen number using a '`search`', showing that, with two processes, the chosen number can easily reach ten (or any other natural). The problem happens when a process chooses a number while the other process is in its critical region. A process only zeroes its chosen number *after* leaving the critical region. It works as follows: process '`0`', with a chosen number of 2, is in its critical region; process '`1`' chooses 3 as its number; when this process is in its critical region, process '`0`' gets another number, which is 4, and so on.

```
search [1] in BAKERY : initial-conf =>*
    < S:Soup,{PR:PreRecord,n = (0 |-> 10 +++ 1 |-> I:Int)} > .

...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 2 +++ 1 |-> 1)} >
```

```
< (prc(0, waiting(1)) prc(1, rem)),
  {n = (0 |-> 2 +++ 1 |-> 0)} >
...
< (prc(0, crit) prc(1, choosing(-1, 2))),
  {n = (0 |-> 2 +++ 1 |-> 0)} >
...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 4 +++ 1 |-> 3)} >
...
< (prc(0, crit) prc(1, choosing(-1, 4))),
  {n = (0 |-> 4 +++ 1 |-> 0)} >
...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 6 +++ 1 |-> 5)} >
...
< (prc(0, crit) prc(1, choosing(-1, 6))),
   {n = (0 |-> 6 +++ 1 |-> 0)} >
...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 8 +++ 1 |-> 7)} >
...
< (prc(0, crit) prc(1, choosing(-1, 8))),
  {n = (0 |-> 8 +++ 1 |-> 0)} >
...
< (prc(0, choosing(1, -1)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, choosing(0, 9)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, choosing(-1, 9)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, waiting(0)) prc(1, waiting(0))),
  {n = (0 |-> 10 +++ 1 |-> 9)} >
```

Let us naively modify the algorithm so that the chosen number is incremented modulo, say, 2267. The algorithm fails for the same reason: chosen numbers gets increasingly high and, using arithmetic modulo 2267, they will eventually be zero, a number that is obviously smaller than all other numbers, as the following search for a race condition shows:

```
search [1] in BAKERY : initial-conf =>*
    < (prc(0, crit) prc(1, crit)) ::: 'Soup,{PR:PreRecord} > .

Solution 1 (state 183534)
states: 183535  rewrites: 5607219 in 34110ms cpu
    (34130ms real) (164386 rewrites/second)
PR:PreRecord --> ch = (0 |-> 0 +++ 1 |-> 0),
                      n = (0 |-> 2266 +++ 1 |-> 0)
```

```
...
< (prc(0, try) prc(1, choosing(-1, 2264))) ,
  {n = (0 |-> 0 +++ 1 |-> 0)} >
< (prc(0, choosing(1, -1)) prc(1, choosing(-1, 2264))),
  {n = (0 |-> 0 +++ 1 |-> 0)} >
< (prc(0, choosing(1, -1)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 2265)} >
< (prc(0, choosing(0, 2265)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 2265)} >
< (prc(0, choosing(-1, 2265)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 2265)} >
< (prc(0, waiting(0)) prc(1, waiting(0))),
  {n = (0 |-> 2266 +++ 1 |-> 2265)} >
< (prc(0, waiting(1)) prc(1, waiting(0))),
  {n = (0 |-> 2266 +++ 1 |-> 2265)} >
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 2266 +++ 1 |-> 2265)} >
< (prc(0, waiting(1)) prc(1, rem)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, rem)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, try)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, choosing(1, -1))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, choosing(0, -1))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, choosing(-1, 2266))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, waiting(0))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, crit)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
```

In order to make this algorithm amenable to verification, we must create an abstraction that captures the essence of the algorithm, but does not have the infinite number of states of the original. The solution follows the ideas described in [46], in which a two-process abstraction is defined and proved to correctly simulate the original specification.

The key to find the correct abstraction in this case is to realize that the actual *absolute value* of the chosen number is not important, but its *relative value* with regard to the other numbers.

We begin with the following two equations: a process changes its number to zero after leaving the critical zone, so the number chosen by the other process in this case does not need to grow indefinitely: choosing number one is sufficient.

```
ceq (< S:Soup, { n = (0 |-> 0 +++ 1 |-> I), PR } >)
  = < S:Soup, { n = (0 |-> 0 +++ 1 |-> 1), PR } >
```

```
if I > 1 .

ceq (< S:Soup, { n = (0 |-> I +++ 1 |-> 0), PR } >)
  = < S:Soup, { n = (0 |-> 1 +++ 1 |-> 0), PR } >
if I > 1 .
```

Next, the following equations keep the chosen numbers of both processes from growing indefinitely, while keeping their relative values.

```
ceq (< S:Soup, { n = (0 |-> I +++ 1 |-> I'), PR } >)
  = < S:Soup, { n = (0 |-> 2 +++ 1 |-> 1), PR } >
if (I' < I) /\ not (I' == 1 and I == 2) .

ceq (< S:Soup, { n = (0 |-> I +++ 1 |-> I'), PR } >)
  = (< S:Soup, { n = (0 |-> 1 +++ 1 |-> 1), PR } >)
if not (I' < I) /\ not (I' == 1 and I == 1) .
```

With these abstractions we may now try a search for a race condition.

```
rewrites: 4195 in 61ms cpu (61ms real) (67671 rewrites/second)
search in CHECK :
   initial-conf =>* <(prc(0,crit)prc(1,crit))::: 'Soup,
   {PR:PreRecord}> .

No solution.
```

Also, both processes eventually reach their critical region, according to the results of the two searches below:

```
rewrites: 3463 in 36ms cpu (36ms real) (93609 rewrites/second)
search in CHECK :
   initial-conf =>* <(prc(0,crit)prc(1,St:St))::: 'Soup,
   {PR:PreRecord}> .

Solution 1
PR:PreRecord <- ch =(0 |-> 0 +++ 1 |-> 0),
                n =(0 |-> 1 +++ 1 |-> 1);
St:St <- waiting(0)
rewrites: 3376 in 26ms cpu (26ms real) (125055 rewrites/second)

search in CHECK :
   initial-conf =>* <(prc(1,crit)prc(0,St:St))::: 'Soup,
   {PR:PreRecord}> .

Solution 1
PR:PreRecord <- ch =(0 |-> 0 +++ 1 |-> 0),
                n =(0 |-> 2 +++ 1 |-> 1);
St:St <- waiting(0)
Bye.
```

E.4 Leader election on an asynchronous ring

This Section specifies the algorithm for *leader election* on an unidirectional, asynchronous ring. It is used as an example of a specification that uses the message-passing model and provides us with more complex model checking examples. The intuitive idea behind this algorithm is to elect as leader the process that has the highest pid of all processes in the ring. Each process forwards its own pid to its neighbor. A process, upon receiving a pid that is greater than its own, forwards it to its neighbor. The greatest pid will eventually circle the ring arriving back at its origin. When a process receives its own pid from a neighbor, it knows it is the leader. It may initiate now, for example, a broadcast announcing the leader election.

Let us begin the formal description of the algorithm by defining the format of the messages. It contains as first argument a pid and as the second argument the destination of the message. There is no need to keep track of the source of the message, as we are dealing with a known network topology.

```
Msg ::= m Int to Int .
```

The ring network is modelled in this specification by having each process knowing the pid for its neighbor. Only one neighbor is known, hence communication in this specification is made in only one direction throughout the ring.

```
Proc ::= prc (Int, Int', St) .
```

As usual, we show the states of a process, while explaining their meaning on the subsequent transitions.

```
St ::= start
     | waiting
     | leader .
```

At the beginning of the algorithm, each process sends its pid to its neighbor.

```
prc (Int, Int', start) : Soup -->
               prc (Int, Int', waiting) (m Int to Int') .
```

When a process receives a message from a neighbor, it compares its pid $\mathtt{i}$ with the pid $\mathtt{i}'$ received from its neighbor. If $\mathtt{i}' > \mathtt{i}$, it forwards the message to its own neighbor.

```
   Int'' > Int
-- -----------------------------------------------------
prc (Int, Int', waiting) (m Int'' to Int) : Soup -->
            prc (Int, Int', waiting) (m Int'' to Int') .
```

If $\mathtt{i}' < \mathtt{i}$, it removes the message from the ring.

```
   Int'' < Int
-- ------------------------------------------------------
prc (Int, Int', waiting) (m Int'' to Int) : Soup -->
            prc (Int, Int', waiting) .
```

When $i' = i$ the process know it is the leader.

```
   Int'' == Int
-- -------------------------------------------------------
prc (Int, Int', waiting) (m Int'' to Int) : Soup -->
                                prc (Int, Int', leader) .
```

In order to verify the correctness of the specification, let us make some verifications using Maude's model checker on a configuration with four processes. We begin by creating an operation '`leaders(S)`' that computes the number of leaders in a soup S.

```
op leaders : Soup -> Int .

eq leaders (S S') = leaders (S) + leaders (S') .
eq leaders (prc (I, I', leader)) = 1 .
eq leaders (prc (I, I', waiting)) = 0 .
eq leaders (prc (I, I', start)) = 0 .
eq leaders (m I to I') = 0 .
```

The proposition '`one-leader`' holds when there is exactly one leader on the configuration, while '`no-leader`' holds when there is no leader on the configuration.

```
op one-leader : -> Prop .
op no-leader : -> Prop .

eq < S ::: 'Soup, R > |= one-leader = (leaders (S) == 1) .
eq < S ::: 'Soup, R > |= no-leader = (leaders (S) == 0) .
```

We may now model check our first formula: in all executions of the specification, there is always one leader.

```
rewrites: 7339322 in 56444ms cpu (56447ms real)
          (130027 rewrites/second)
reduce in CHECK :
  modelCheck(init, <> [] one-leader)
result Bool :
  true
```

There is no execution in which a leader is not elected.

```
rewrites: 7204804 in 56551ms cpu (57278ms real)
          (127402 rewrites/second)
reduce in CHECK :
  modelCheck(init, ~ [] no-leader)
result Bool :
  true
```

In all executions, there is no leader until a leader is selected.

```
rewrites: 7340679 in 57867ms cpu (58915ms real)
          (126853 rewrites/second)
reduce in CHECK :
  modelCheck(init, [] (no-leader U one-leader))
result Bool :
  true
```

APPENDIX F – Combinatory Logic in Maude

The purpose of this Chapter is to demonstrate that, depending on the characteristics of the specification, the Maude engine can reach rewrite speeds on the order of 10^6 rewrites/second. As an example, we will specify a simple *combinatory logic* interpreter. The literature on combinatory logic is vast, and we opted to follow the description and examples in [9].

We begin by defining the primitive combinators S and K and their semantics.

```
fmod CL is
 sort Exp .

 op S : -> Exp [ctor] .
 op K : -> Exp [ctor] .

 op __ : Exp Exp -> Exp [gather(E e)] .

 vars x y z : Exp .

 eq K x y = x .
 eq S x y z = x z (y z) .
endfm
```

The module 'CL-EXT' extends this very basic set of combinators with Curry's B, C, and I combinators, defined in terms of Shönfinkel's S and K.

```
fmod CL-EXT is including CL .
 op B : -> Exp .  eq B = ((S(K S))K) .
 op C : -> Exp .  eq C = ((S(K((S S)(K K))))((S(K K))S)) .
 op I : -> Exp .  eq I = ((S K)K) .
endfm
```

This combined set is then used to define the positive integers as the following progression shows:

$$
\begin{array}{rcc}
1 & \equiv & ((\mathsf{SB})(\mathsf{KI})) \\
2 & \equiv & ((\mathsf{SB})((\mathsf{SB})(\mathsf{KI}))) \\
3 & \equiv & ((\mathsf{SB})((\mathsf{SB})((\mathsf{SB})(\mathsf{KI})))) \\
 & \cdots &
\end{array}
$$

Module '`CL-NATURALS`' implements this idea with the operator '`$(n)`' that converts the natural n into its equivalent expression. We also define some additional operators that implement addition, multiplication, and exponentiation: respectively, '`pl`', '`ti`', and '`ex`'.

```
fmod CL-NATURALS is including CL-EXT .
 op $ : Nat -> Exp .

 var n : Nat .

 eq $(s(n)) = (S B) $(n) .
 eq $(0) = (K I) .

 op pl : -> Exp .
 op ti : -> Exp .
 op ex : -> Exp .

 eq pl = ((C I) (S B)) .
 eq ti = ((B((C C)(K I)))((C B) pl)) .
 eq ex = (C((B(C((C C)((S B)(K I))))) ti)) .
endfm
```

With '`CL-NATURALS`' it is possible to calculate, for example, $1 + 1$.

```
Maude> red pl $(1) $(1) .
reduce in CL-NATURALS : pl $(1) $(1) .
rewrites: 28 in 0ms cpu (0ms real) (~ rewrites/second)
result Exp: S (S (K S) K) (S (S (K S) K) (K (S K K)))
```

We would like to find a way of converting back this sequence into its equivalent number, for obvious reasons. The key for this conversion is to notice that numbers, when seen as combinators, have an interesting property: if ε_1 and ε_2 are expressions, and $\hat{n}$ is the combinator expression equivalent to number n, then, $\hat{1}\varepsilon_1\varepsilon_2 = \varepsilon_1\varepsilon_2, \hat{2}\varepsilon_1\varepsilon_2 = \varepsilon_1\varepsilon_1\varepsilon_2, \hat{3}\varepsilon_1\varepsilon_2 = \varepsilon_1\varepsilon_1\varepsilon_1\varepsilon_2, \ldots$ Module '`NATURALS-CL`' implements this idea.

```
fmod NATURALS-CL is including CL-NATURALS .
 vars x y : Exp .

 ops eqv eqv-aux : Exp -> Nat .

 ops i j : -> Exp .

 eq eqv (x) = eqv-aux (x i j) .
 eq eqv-aux (x y) = eqv-aux(x) + eqv-aux(y) .
 eq eqv-aux (i) = 1 .
 eq eqv-aux (j) = 0 .
endfm
```

Finally, the following reduction achieves the announced order of a million rewrites per second. This is certainly caused by the huge repetitive patterns of S and K constants on the juxtaposition operator, facilitating the matching algorithm implemented by the Maude interpreter.

```
Maude> reduce in NATURALS-CL : eqv(ti $(500) $(500)) .
reduce in NATURALS-CL : eqv(ti $(500) $(500)) .
rewrites: 2506069 in 2178ms cpu (2184ms real)
   (1150275 rewrites/second)
result NzNat: 250000
```