Executable Structural Operational Semantics in Maude[*]

Alberto Verdejo and Narciso Martí-Oliet

Technical Report 134-03

Departamento de Sistemas Informáticos y Programación,
Universidad Complutense de Madrid

September 3, 2003

Abstract

This paper describes in detail how to bridge the gap between theory and practice when implementing in Maude structural operational semantics described in rewriting logic, where transitions become rewrites and inference rules become conditional rewrite rules with rewrites in the conditions, as made possible by the new features in Maude 2.0. We validate this technique using it in several case studies: a functional language *Fpl* (evaluation and computation semantics, including an abstract machine), imperative languages *WhileL* (evaluation and computation semantics) and *GuardL* with nondeterminism (computation semantics), Kahn's functional language Mini-ML (evaluation or natural semantics), Milner's CCS (with strong and weak transitions), and Full LOTOS (including ACT ONE data type specifications). In addition, on top of CCS we develop an implementation of the Hennessy-Milner modal logic for describing local capabilities of processes, and for LOTOS we build an entire tool where Full LOTOS specifications can be entered and executed (without user knowledge of the underlying implementation of the semantics). We also compare this method based on transitions as rewrites with another one based on transitions as judgements.
Keywords: Rewriting logic, Maude 2.0, executability, structural operational semantics, metalanguage, CCS, LOTOS, ACT ONE.

1 Introduction

In the context of proposing rewriting logic as a logical and semantic framework, the paper [48] illustrated several different ways of mapping inference systems into rewriting logic. A very general possibility is to map an inference rule of the form

$$\frac{S_1 \ldots S_n}{S_0}$$

into a rewrite rule of the form $S_1 \ldots S_n \longrightarrow S_0$ that rewrites *multisets* of judgements S_i. This mapping is correct from an abstract point of view, as justified in [48], but thinking in terms of executability of the rewrite rules, it is more appropriate to consider rewrite rules of the form $S_0 \longrightarrow S_1 \ldots S_n$ that still rewrite multisets of judgements but go from the conclusion to the premises, so that rewriting with these rules corresponds to searching for a proof in a bottom-up way. Again this mapping is correct, and in both cases the intuitive idea is that the rewriting relation corresponds to the horizontal bar separating conclusion from premises in the typical textbook presentation of inference rules. We call this method *transitions as judgements*.

[*]Research supported by CICYT projects *Desarrollo Formal de Sistemas Basados en Agentes Móviles* (TIC2000–0701–C02–01) and *Métodos y Herramientas Lógicas para el Diseño y la Verificación de Software Multiparadigma* (TIC2002–01167).

These mappings can be applied to a wide variety of inference systems, as explained in [48], including sequent systems for logics and also structural operational semantics definitions for languages. However, in the operational semantics case, judgements S_i typically have the form of some kind of transition $P_i \to Q_i$ between states so that it makes sense to consider the possibility of mapping directly this transition relation between states to a rewriting relation between terms representing the states. When thinking this way, an inference rule of the form

$$\frac{P_1 \to Q_1 \quad \dots \quad P_n \to Q_n}{P_0 \to Q_0}$$

becomes a *conditional* rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \textit{if} \quad P_1 \longrightarrow Q_1 \wedge \dots \wedge P_n \longrightarrow Q_n,$$

where the condition includes rewrites. In this way the semantic rules become (conditional) rewrite rules, where the transition in the conclusion becomes the main rewrite of the rule, and the transitions in the premises become rewrite conditions. We call this method *transitions as rewrites*.

Rules of this form were already considered by Meseguer in the seminal paper [50] on rewriting logic. At the logical level, this mapping is again correct, but one must be careful to take into account in the mapping additional information appearing in the transitions of the operational semantics. For example, in structural operational semantics for process algebras it is essential for the transitions to have some labelling information that provides the mechanism for synchronization. How to solve these details in the particular case of Milner's CCS [52] was already shown in [48]. Moreover, the papers [11, 20] showed the good properties of this semantic mapping for CCS.

The recent availability of the rewriting logic language Maude 2.0 [15, 16, 17] has made it possible to put into practice the approach based on transitions as rewrites, because Maude 2.0 allows indeed conditional rules with rewrites in the conditions, where those rewrites are solved at execution time by means of a built-in search mechanism. Thus, we undertook the project of carefully implementing in a *fully executable* way the CCS operational semantics in order to practically assess the ideas summarized above that theoretically were elegant and correct. CCS was taken only as a first example; the desired solutions in our search of executability would should be general enough to handle many other operational semantics definitions, considering the approach of transitions as rewrites and using conditional rules with rewrites in the conditions.

To validate this approach we have thus far considered several different operational semantics for programming languages. On the one hand we have implemented all the evaluation (big step) and computation (small step) semantics presented by Hennessy in [36] for functional and imperative programming languages, including several variants like call-by-value versus call-by-name, or using substitutions versus extending the environment. This paper describes all of them in full detail. Another functional language whose semantics we have implemented is Kahn's Mini-ML [45]; the letrec syntactic construct in this language requires a special treatment that deserves consideration.

On the other hand, in addition to the operational semantics of the CCS process algebra, we have implemented in Maude a symbolic semantics for LOTOS [44], following this technique of transitions as rewrites. Moreover, in the case of these process algebras we show how the implementation of the semantics can be used to develop formal analysis tools. In the CCS case we integrate it with an implementation of the Hennessy-Milner modal logic [38] for describing local capabilities of processes; and in the LOTOS case, we integrate it with a translation of ACT ONE [27] data type specifications into functional modules in Maude, building an entire tool where Full LOTOS specifications can be entered and executed (without user knowledge of the underlying implementation of the semantics).

In our opinion, the approach based on transitions as rewrites is really simpler than the one based on transitions as judgements, because it is closer to the mathematical textbook presentation of the operational semantics and in general requires less auxiliary structures or operations. However, there is still the need to bridge some gaps between theory and practice, and in this case the new `frozen` attribute available in Maude 2.0 has also played an important role, as described in detail in Section 6.2.

The declaration of an operator as frozen forbids rewriting its arguments, thus providing another way of controlling the rewriting process.

An important and very outstanding characteristic of all our implementations provided by the Maude language and system is the integration in the same framework of all the specification levels necessary to implement in detail the semantics of a given language. First, writing the grammar for the (abstract) syntax of the language corresponds to defining the algebraic signature in the corresponding algebraic specification. Then, all the data and corresponding operations necessary for the implementation are described by means of equational specifications. On top of them, the (dynamic) semantics of the language is defined by means of rewrite rules, as we have described above. Although we do not treat this in this paper, it is also possible to describe the static semantics of a language (like type checking, for example) using these techniques, but in this case the approach based on transitions as judgements may be more appropriate. In addition, if more control is necessary one can user reflection to go up to the metalevel, as we do for instance in Section 6.4 to define the semantics of a modal logic that requires considering all possible rewrites (or transitions). The integration of all these aspects is precisely what allows our development of a semantics for Full LOTOS, where the ACT ONE data type specifications are translated into functional Maude modules. Furthermore, the metalanguage features of Maude allow us to build in a fully integrated way a tool for Full LOTOS that includes input/output, parsing (taking into account the user-definable syntax of ACT ONE specifications), execution, pretty-printing, etc. in such a way that the Maude definition is hidden from the user that only needs to know about the Full LOTOS specifications that are to be executed.

Concerning the implementation of the operational semantics, the methods that we describe in this paper assume that we are given a correct structural operational semantics for a language. Of course, it is also possible to develop such an implementation precisely with the purpose of prototyping a proposed semantics and then modify it according to the information obtained by the tests in the execution. Assuming the semantics is given, the first step consists in identifying the structure necessary to build the lefthand and righthand sides of the rewrites. Although this may be very simple in some cases, in our examples we can see that usually there is *additional structure* that has to be taken into account, like environments in the case of functional languages that we put in the lefthand side, or synchronization labels in the case of process algebras that we put in the righthand side.

Having done this, in general, translating the semantics rules into rewrite rules is a quite systematic process. However, one has to make sure that those rules are executable and, moreover, that they execute as one expects. For example, in the CCS example, we will see the use the `frozen` attribute to ensure that rewriting only takes place at the top of a state, and that rewrite conditions do not give rise to non-terminating executions. As another example, the mathematical rule for the letrec construct of Mini-ML is not executable, because there is an existential variable in the condition that appears both in the lefthand and righthand sides of a rewrite and that in principle one does not know how to instantiate.

All the examples in this paper provide full details about our treatment of these semantics. After a brief review of the main features of Maude 2.0 in Section 2, the following sections develop the details of several case studies: a functional language *Fpl* (evaluation and computation semantics with variants, as well as an abstract machine), an imperative language *WhileL* (evaluation and computation semantics), an imperative nondeterministic language *GuardL* (computation semantics), Kahn's functional language Mini-ML (evaluation or natural semantics), Milner's CCS (with strong and weak transitions, plus the Hennessy-Milner logic), and Full LOTOS (including ACT ONE data type specifications and a complete tool). Section 8 summarizes the transitions as judgements approach and compares it with the transitions as rewrites. Section 9 reviews some related work and Section 10 concludes the paper describing some ideas for future work.

Some results in this paper were previously reported in the conference papers [72, 68], and in the PhD. thesis [70].

2 Rewriting logic and Maude

Rewriting logic was introduced by Meseguer [50] as a unified model of concurrency in which several well-known models of concurrent systems can be represented in a common framework. Since then much work has been done on the use of rewriting logic as a logical and semantic framework [48, 49], in which many different logics, models of computation, and a wide range of languages, including formal specification languages, can be represented, given a precise semantics, and executed. Among the advantages of rewriting logic, we may emphasize the following:

- It is a simple formalism, with only a few rules of deduction that are easy to understand and justify.
- It is very flexible and expressive, capable of representing change in systems with very different structure.
- It allows user-definable syntax, with complete freedom to choose the operators and structural properties appropriate for each problem.
- It is intrinsically concurrent, representing concurrent change and supporting reasoning about such change.
- It supports modelling of concurrent object-oriented systems in a simple and direct way.
- It has initial models, that can be intuitively understood as providing "no junk" and "no confusion."
- It is realizable in the wide spectrum logical language Maude, supporting executable specification and programming.

Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation. We use in this paper Maude 2.0 [15, 16, 17], a new version with greater generality and expressiveness; in particular, Maude 2.0 allows rewrite conditions which are essential for the implementation of the semantic definitions we are going to present.

In rewriting logic and Maude the data on the one hand and the state of a system on the other are both formally specified as an algebraic data type by means of an equational specification. In this kind of specifications we can define new types (by means of the keyword `sort`); subtype relations between types (`subsort`); operations (`op`) for building values of these types, giving the types of their arguments and result, and which may have attributes as being associative (`assoc`) or commutative (`comm`), for example; and equations (`eq`) that identify terms built with these operators. The following *functional module* (with syntax `fmod...endfm`) defines the natural numbers with an addition operation:

```
fmod NATURAL-NUMBERS is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [assoc comm].
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + s(M) = s(s(N + M)) .
endfm
```

Equations are assumed to be confluent and terminating, so that we can use the equations to reduce a term t to a unique, canonical form t' that is equivalent to t (they represent the same value).

Maude uses a very expressive version of equational logic, namely *membership equational logic* [2, 51], that (in addition to all the above) allows the statement of *membership assertions* (`mb`) characterizing the elements of a sort. For example, we can extend the `NATURAL-NUMBERS` module with the following two memberships

```
  mb 0 : Even .
  mb s(s(E:Even)) : Even .
```

defining a subsort Even of even natural numbers. Notice the on-the-fly declaration for the variable E of sort Even. In Maude 2.0 a variable is an identifier composed of a name, a colon, and a sort name; in this way, variables do not have to be declared in variable declarations, although such declarations are still allowed for convenience.

Membership equational logic also has a notion of (implicit) error supersorts called *kinds*, which in Maude are not explicitly declared, but are instead represented as sort names between square brackets. Using kinds, we can declare partial operations (at the level of sorts), like for example the following integer division operation on natural numbers:

```
  op _div_ : Nat Nat -> [Nat] .
```

Notice that this is not at all the only possible way of treating partiality in membership equational logic. For example we could also define a subsort NzNat of non-zero natural numbers and then declare _div_ as a total operation

```
  op _div_ : Nat NzNat -> Nat .
```

The *dynamic* behaviour of a distributed system is specified by rewrite rules of the form $t \longrightarrow t'$, that describe the local, concurrent transitions of the system. That is, when a part of a system matches the pattern t, it can be transformed into the corresponding instance of the pattern t'. Rewrite rules are included in *system modules* (with syntax mod...endm). For example, the following module defines non-deterministic natural numbers and non-deterministic choice. A module can import, or include, the definitions of another module by means of the keyword inc (short for including).

```
mod NONDETERMINISTIC-NATURAL-NUMBERS is
  inc NATURAL-NUMBERS .
  sort NdNat .
  subsort Nat < NdNat .
  op _?_ : NdNat NdNat -> NdNat [assoc comm].
  var N : Nat .  var ND : NdNat .
  rl [choice] : N ? ND => N .
endm
```

A multiset of natural numbers is regarded as a non-deterministic natural number of sort NdNat, that is, a number that could be any among those in the multiset. The operation _?_ denotes the union of non-deterministic natural numbers, which is associative and commutative, and the choice rule provides non-deterministic choice.

Rewrite rules can take the most general possible form in the variant of rewriting logic built on top of membership equational logic, that is, they can be of the form

$$t \rightarrow t' \;\; \mathit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \rightarrow q_k)$$

with no restriction on which new variables may appear in the righthand side or in the condition. Conditions in rules are formed by an associative conjunction connective /\, allowing equations (both ordinary equations t = t', and matching equations t := t' where new variables occurring in t become instantiated by matching [15, 16]), memberships (t : s), and rewrites (t => t') as conditions. In that full generality the execution of a system module will require *strategies* that control at the metalevel the instantiation of the extra variables in the condition and in the righthand side. However, a quite general class of system modules, called *admissible modules*, are executable by Maude 2.0's default interpreter. Essentially, the admissibility requirement ensures that all the extra variables will eventually become instantiated by matching [15].

When executing a conditional rule, the satisfaction of all its conditions is attempted sequentially from left to right; but notice that, besides the fact that many matches for the equational conditions may be possible due to the presence of equational axioms, we also have to deal with the fact that solving rewrite conditions requires *search*, including searching for new solutions when previous ones fail to satisfy subsequent conditions. Therefore, the default interpreter supports search computations. The `search` command looks for all the rewrites of a given term that match a given pattern satisfying some condition (we will see some examples in Section 3.2).

Another Maude 2.0 feature which is very important for our intended semantics applications is the `frozen` attribute [16]. When an operation is declared as frozen, its arguments cannot be rewritten by rules (it is also possible to declare operations with only some arguments frozen, but we will not make use of this generality). This is important in situations where the rewriting process should only happen at the top, like in many operational semantics for process algebras, like CCS, as we will see in Section 6.2; however, there are more reasons for using the `frozen` attribute, related in general to avoiding situations of non-termination in the execution of rewrite conditions, as we will explain in some detail in Section 6.2.

Maude should be viewed as a *metalanguage* [14] in which the syntax and semantics of computational models and languages can be formally defined, and in which entire *environments* for such languages can be built (including parsers, execution environments, pretty printing, and input/output). We will see how an environment of this kind has been built for LOTOS in Section 7.4.

Reflection is the main feature to achieve these powerful metalanguage functionalities. Rewriting logic is reflective [12, 18], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$, and we then have the following equivalence:

$$\mathcal{R} \vdash t \longrightarrow t' \;\;\Leftrightarrow\;\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Intuitively, this means that we can work with theories as *data* at the metalevel, combining and manipulating them, and controlling the rewriting process.

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`, where Maude terms are reified as elements of a data type `Term`, Maude modules are reified as terms in a data type `Module`, the process of reducing a term to normal form is reified by a function `metaReduce`, and the process of applying a rule of a system module to a subject term is reified by a function `metaApply` [16]. These basic operations can be combined to build *strategies* [12] that control the process of rewriting.

Search is also reified at the metalevel by means of the operation `metaSearch` (used in Section 6.4), which receives as arguments the metarepresented module to work in, the starting term for search, the pattern to search for, a side condition, the kind of search (which may be `'*` for zero or more rewrites, `'+` for one or more rewrites, and `'!` for only matching normal forms), the depth of search, and the required solution number which is used to index all possible solutions. It returns the term matching the pattern, its type, and the substitution produced by the match.

For creating an environment for a language using Maude, we need generic syntax definition, meta-parsing, and meta-pretty printing capabilities that can deal with expressions in any language, including languages like Maude itself whose modules have user-definable syntax. And we need a general facility for input/output that can be customized for each language of interest. Section 7.4 explains how all this is done in Maude thanks to its reflective design, in our application of these techniques to the development of a tool for Full LOTOS.

3 The functional language *Fpl*

We begin our description of how to implement structural operational semantics in Maude with a simple functional language. *Fpl* (*Functional Programming Language* [36]) is a language with arithmetic and

1. Syntactic categories

p	$\in$	$Prog$	op	$\in$	Op	x	$\in$	Var
D	$\in$	Dec	bop	$\in$	BOp	bx	$\in$	$BVar$
e	$\in$	Exp	n	$\in$	Num	F	$\in$	$FunVar$
be	$\in$	$BExp$						

2. Definitions

$$\begin{array}{rcl}
p & ::= & \langle e, D\rangle \\
D & ::= & F(x_1,\ldots,x_k) \Longleftarrow e \mid F(x_1,\ldots,x_k) \Longleftarrow e, D \\
op & ::= & + \mid - \mid * \\
bop & ::= & And \mid Or \\
e & ::= & n \mid x \mid e'\ op\ e'' \mid \mathsf{If}\ be\ \mathsf{Then}\ e'\ \mathsf{Else}\ e'' \mid \mathsf{let}\ x = e'\ \mathsf{in}\ e'' \mid F(e_1,\ldots,e_k) \\
be & ::= & bx \mid \mathsf{T} \mid \mathsf{F} \mid be'\ bop\ be'' \mid \mathsf{Not}\ be' \mid \mathsf{Equal}(e,e')
\end{array}$$

Figure 1: Abstract syntax for *Fpl*.

Boolean expressions, if-then-else, local variable declarations (let), function declarations defined by the user (with the possibility of mutual recursion), and function calls.

In this paper we describe the implementation of three different semantics for *Fpl*: a quite abstract evaluation semantics, a more detailed computation semantics, and an even more concrete semantics which uses an *abstract machine*. A different functional language (Mini-ML) is implemented later in Section 5.

3.1 Functional syntax definition

The abstract syntax of *Fpl*, with an obvious intuitive meaning, is described in Figure 1. A *program* consists of an expression together with a declaration, $\langle e, D\rangle$. Intuitively, D supplies the definitions for all the function names in e. By using membership axioms we could define when a program $\langle e, D\rangle$ is correct, that is, when all the functions used in e are defined in D, but we have not done so here in order to simplify the presentation.

This syntax is implemented in the following functional module `FPL-SYNTAX`. Note that the signature structure corresponds to the grammar structure defined by the syntax of the language in Figure 1 (the `prec` attribute is used to associate precedence values to the different operators, so that parentheses are not necessary to disambiguate terms [16]). In addition, as can be expected in a really executable semantics, we had to fill in the details that in the textbook presentation are left out, like the definition of the natural numbers in this case.

```
fmod FPL-SYNTAX is
  protecting QID .

  sorts Var Num Op Exp BVar Boolean BOp BExp FunVar
        VarList NumList ExpList Prog Dec .

  op V : Qid -> Var .
  subsort Var < Exp .
  subsort Num < Exp .

  op BV : Qid -> BVar .
  subsort BVar < BExp .
  subsort Boolean < BExp .

  op FV : Qid -> FunVar .

  ops + - * : -> Op .

  op 0 : -> Num .
  op s : Num -> Num .
  subsort Exp < ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc prec 30] .

  op ___ : Exp Op Exp -> Exp [prec 20] .
  op If_Then_Else_ : BExp Exp Exp -> Exp [prec 25] .
  op let_=_in_ : Var Exp Exp -> Exp [prec 25] .
  op _`(_`) : FunVar ExpList -> Exp [prec 15] .

  ops T F : -> Boolean .
  ops And Or : -> BOp .
  op ___ : BExp BOp BExp -> BExp [prec 20] .
  op Not_ : BExp -> BExp [prec 15] .
  op Equal : Exp Exp -> BExp .

  subsort Var < VarList .
  op _,_ : VarList VarList -> VarList [assoc prec 30] .
  subsort VarList < ExpList .

  subsort Num < NumList .
  op _,_ : NumList NumList -> NumList [assoc prec 30] .
  subsort NumList < ExpList .

  op <_,_> : Exp Dec -> Prog .
  op nil : -> Dec .
  op _`(_`)<=_ : FunVar VarList Exp -> Dec [prec 30] .
  op _&_ : Dec Dec -> Dec [assoc comm id: nil prec 40] .

  op exDec1 : -> Dec .
  eq exDec1 =
    FV('Fac)(V('x)) <= If Equal(V('x),0) Then s(0)
                       Else V('x) * FV('Fac)(V('x) - s(0)) &
    FV('Rem)(V('x) , V('y)) <= If Equal(V('x),V('y)) Then 0
                             Else If Equal(V('y) - V('x), 0) Then V('y)
                                  Else FV('Rem)(V('x) , V('y) - V('x)) &
    FV('Double)(V('x)) <= V('x) + V('x) .
endfm
```

We use the predefined quoted identifiers, of sort `Qid`, for representing variable identifiers in the language *Fpl*. Instead of declaring this sort as a subsort of `Var`, since `Qid` is also used to represent Boolean variables, we have constructors `V` and `BV` that transform the `Qid`s to values of sorts `Var` and `BVar`, respectively. As arithmetic constants we use the natural numbers in Peano notation, with constructors `0` and `s`.

In addition to the complete syntax of *Fpl*, the above module includes a constant `exDec1`, with a set of function declarations that we will use later on.

The abstract syntax of *Fpl* in Figure 1 is common for the three semantic definitions presented in [36], and that we are going to see in the following sections. However, in order to make easier the representation in Maude 2.0 of these different semantics, some changes will be done in module `FPL-SYNTAX`. These changes could have been done from the beginning, but we prefer to have different versions in order to point out the (small) differences.

We define in another functional module `AP` an operation *Ap* for the application of a binary operator to two already evaluated arguments. Again, this module supplies some details that are usually left out in a textbook presentation of the semantics. A third functional module `ENV` is used to define *environments* that associate values to variables, either arithmetic or Boolean. These two modules are independent of the concrete representation of the semantics.

```
fmod AP is
  protecting FPL-SYNTAX .

  op Ap : Op Num Num -> Num .
  vars n n' : Num .

  eq Ap(+, 0, n) = n .
  eq Ap(+, s(n), n') = s(Ap(+, n, n')) .
  eq Ap(*, 0, n) = 0 .
  eq Ap(*, s(n), n') = Ap(+, n', Ap(*, n, n')) .
  eq Ap(-, 0, n) = 0 .
  eq Ap(-, s(n), 0) = s(n) .
  eq Ap(-, s(n), s(n')) = Ap(-, n, n') .

  op Ap : BOp Boolean Boolean -> Boolean .

  var bv bv' : Boolean .

  eq Ap(And, T, bv) = bv .
  eq Ap(And, F, bv) = F .
  eq Ap(Or, T, bv) = T .
  eq Ap(Or, F, bv) = bv .
endfm

fmod ENV is
  protecting FPL-SYNTAX .

  sorts Value Variable .
  subsorts Num Boolean < Value .
  subsorts Var BVar < Variable .

  sort ENV .
  op mt : -> ENV .
  op _=_ : Variable Value -> ENV [prec 20] .
  op __ : ENV ENV -> ENV [assoc id: mt prec 30] .
  op _(_) : ENV Variable -> Value .
  op _[_/_] : ENV Value Variable -> ENV [prec 35] .
  op remove : ENV Variable -> ENV .

  vars X X' : Variable .  var V : Value .  var rho : ENV .

  eq (X = V rho)(X') = if X == X' then V else rho(X') fi .
  eq rho [V / X] = remove(rho, X) X = V .
  eq remove(mt, X) = mt .
  eq remove(X = V rho, X') = if X == X' then rho else X = V remove(rho,X') fi .
endfm
```

Operations `mt`, `_=_` and `__` (in the module `ENV`) are used to build empty environments, singleton environments, and union of environments, respectively. The operation `_(_)` is used to look up the value associated to a variable in an environment, and it is defined recursively by means of an equation. The operation `_[_/_]` is used to modify the binding between a variable and a value in an environment, and it is defined by means of the auxiliary operation `remove` that eliminates a given variable from an environment.

3.2 Evaluation semantics

The evaluation semantics for *Fpl* is given by means of two relations: $\Longrightarrow_A$ and $\Longrightarrow_B$, corresponding, respectively, to arithmetic and Boolean expressions. For evaluating an arithmetic expression e we need an environment ρ which assigns concrete values to the variables occurring in e, and a set of declarations D giving a context for the function names in e. Thus, judgements in this semantics will have the form $D, \rho \vdash e \Longrightarrow_A v$. The same happens with Boolean expressions since, although function calls are only arithmetic expressions, these expressions can also be used to build Boolean expressions

$$\text{CR}\quad \frac{}{D,\rho \vdash \mathbf{n} \Longrightarrow_A \mathbf{n}} \qquad\qquad \text{VarR}\quad \frac{}{D,\rho \vdash x \Longrightarrow_A \rho(x)}$$

$$\text{OpR}\quad \frac{D,\rho \vdash e \Longrightarrow_A v \qquad D,\rho \vdash e' \Longrightarrow_A v'}{D,\rho \vdash e\ op\ e' \Longrightarrow_A Ap(op,v,v')}$$

$$\text{IfR}\quad \frac{D,\rho \vdash be \Longrightarrow_B \mathsf{T} \qquad D,\rho \vdash e \Longrightarrow_A v}{D,\rho \vdash \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e' \Longrightarrow_A v} \qquad \frac{D,\rho \vdash be \Longrightarrow_B \mathsf{F} \qquad D,\rho \vdash e' \Longrightarrow_A v'}{D,\rho \vdash \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e' \Longrightarrow_A v'}$$

$$\text{LocR}\quad \frac{D,\rho \vdash e \Longrightarrow_A v \qquad D,\rho[v/x] \vdash e' \Longrightarrow_A v'}{D,\rho \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \Longrightarrow_A v'}$$

$$\text{FunR}\quad \frac{\begin{array}{c} D,\rho \vdash e_i \Longrightarrow_A v_i,\ 1 \leq i \leq k \\ D,\rho[v_1/x_1,\ldots,v_k/x_k] \vdash e \Longrightarrow_A v \end{array}}{D,\rho \vdash F(e_1,\ldots,e_k) \Longrightarrow_A v}\ F(x_1,\ldots,x_k) \Leftarrow e \text{ is in } D$$

Figure 2: Evaluation semantics for *Fpl*, $\Longrightarrow_A$.

$$\text{BCR}\quad \frac{}{D,\rho \vdash \mathsf{T} \Longrightarrow_B \mathsf{T}} \qquad \frac{}{D,\rho \vdash \mathsf{F} \Longrightarrow_B \mathsf{F}} \qquad\qquad \text{BVarR}\quad \frac{}{D,\rho \vdash bx \Longrightarrow_B \rho(bx)}$$

$$\text{BOpR}\quad \frac{\begin{array}{c} D,\rho \vdash be \Longrightarrow_B bv \\ D,\rho \vdash be' \Longrightarrow_B bv' \end{array}}{D,\rho \vdash be\ bop\ be' \Longrightarrow_B Ap(bop,bv,bv')}$$

$$\text{NotR}\quad \frac{D,\rho \vdash be \Longrightarrow_B \mathsf{T}}{D,\rho \vdash \mathsf{Not}\ be \Longrightarrow_B \mathsf{F}} \qquad \frac{D,\rho \vdash be \Longrightarrow_B \mathsf{F}}{D,\rho \vdash \mathsf{Not}\ be \Longrightarrow_B \mathsf{T}}$$

$$\text{EqR}\quad \frac{\begin{array}{c} D,\rho \vdash e \Longrightarrow_A v \\ D,\rho \vdash e' \Longrightarrow_A v \end{array}}{D,\rho \vdash \mathsf{Equal}(e,e') \Longrightarrow_B \mathsf{T}} \qquad \frac{\begin{array}{c} D,\rho \vdash e \Longrightarrow_A v \\ D,\rho \vdash e' \Longrightarrow_A v' \end{array}}{D,\rho \vdash \mathsf{Equal}(e,e') \Longrightarrow_B \mathsf{F}}\ v \neq v'$$

Figure 3: Evaluation semantics for Boolean expressions, $\Longrightarrow_B$.

by means of the Equal operator. Judgements for evaluating Boolean expressions will be of the form $D,\rho \vdash be \Longrightarrow_B bv$.

By definition, we have that $\rho \vdash \langle e, D\rangle \Longrightarrow v$ if and only if $D,\rho \vdash e \Longrightarrow_A v$. The semantic rules for the transition relation $\Longrightarrow_A$ are shown in Figure 2, and the corresponding ones for the transition relation $\Longrightarrow_B$ in Figure 3.

The rule FunR says that for evaluating $F(e_1,\ldots,e_k)$, first all the arguments have to be evaluated, and then the body of the definition of F has to be evaluated, in an environment where the formal parameters have been bound to the values of the corresponding actual parameters. This is the mechanism known as *call-by-value*. Below we will also see the alternative known as *call-by-name*.

The semantics uses the operation *Ap* for applying a binary operator to two arguments, implemented above in the module `AP`. Variable environments and the operation for the modification of their bindings are implemented in the module `ENV`. The next module `EVALUATION` has the rewrite rules representing the evaluation semantics for *Fpl*, both for arithmetic and Boolean expressions.

```
mod EVALUATION is
  protecting AP .  protecting ENV .
```

In order to represent the semantic rules in Maude, first the elements on both sides of the arrow in a judgement have to be represented as terms in Maude. In this semantics, on the left we have a set of declarations, an environment, and an expression. These three elements are represented by a term of sort `Statement`. On the right we can have an arithmetic or Boolean expression, or a list of arithmetic expressions (as we will see in a moment). Notice the use of the sort `Statement` to ensure that both sides of the rewrite rules are going to have a common sort.

```
sort Statement .
subsorts Num Boolean NumList < Statement .
op _,_|-_ : Dec ENV Exp -> Statement [prec 40] .
op _,_|-_ : Dec ENV BExp -> Statement [prec 40] .
op _,_|-_ : Dec ENV ExpList -> Statement [prec 40] .
```

The axioms (semantic rules without premises) are translated as (unconditional) rewrite rules, where the transition in the conclusion simply becomes the rewrite rule. Rules CR and VarR are two examples.

```
vars D D' : Dec .        var rho : ENV .     var n : Num .
var x   : Var .          var bx : BVar .     var v v' : Num .
var bv bv' : Boolean .   var op : Op .       vars e e' : Exp .
vars be be' : BExp .     var bop : BOp .     var F : FunVar .
var el : ExpList .       var xl : VarList .  var vl : NumList .

rl [CR] : D,rho |- n => n .

rl [VarR] : D,rho |- x => rho(x) .
```

The rest of the semantic rules (with premises) are translated to conditional rewrite rules where the main rewrite corresponds to the transition in the conclusion, and the rewrites in the conditions correspond to the transitions in the premises. Conditions are ordered (remember that they are checked sequentially from left to right), and therefore information can flow from one condition to the next; this happens in the rule `LocR` below, where the value of `v` is obtained in the first condition and is later used in the second.

```
crl [OpR] : D,rho |- e op e' => Ap(op,v,v')
         if D,rho |- e  => v  /\  D,rho |- e' => v' .

crl [IfR1] : D,rho |- If be Then e Else e' => v
          if D,rho |- be => T  /\  D,rho |- e  => v .
crl [IfR2] : D,rho |- If be Then e Else e' => v'
          if D,rho |- be => F  /\  D,rho |- e' => v' .

crl [LocR] : D,rho |- let x = e in e' => v'
          if D,rho |- e  => v  /\  D,rho[v / x] |- e' => v' .
```

The rule FunR presents a problem: the number of premises is not fixed, because it depends on the concrete function call that has to be evaluated, specifically on the number of arguments that it has. We solve this problem by considering the list of actual parameters as a new syntactic category, consisting of non-empty lists of arithmetic expressions, and we write a semantic rule that evaluates lists of expressions. The modified rule FunR and the new rule ExpLR for the evaluation of lists of expressions are the following:

$$\text{FunR} \qquad \frac{\begin{array}{c} D,\rho \vdash el \Longrightarrow_A vl \\ D,\rho[vl/xl] \vdash e \Longrightarrow_A v \end{array}}{D,\rho \vdash F(el) \Longrightarrow_A v}\ F(xl) \Leftarrow e \text{ is in } D$$

$$\text{ExpLR} \qquad \frac{D,\rho \vdash e \Longrightarrow_A v \qquad D,\rho \vdash el \Longrightarrow_A vl}{D,\rho \vdash e, el \Longrightarrow_A v, vl}$$

Their representation as rewrite rules is as follows:

```
*** call-by-value
  crl [FunR] : D,rho |- F(el) => v
           if D,rho |- el => vl /\ F(xl)<= e & D' := D /\ D,rho[vl / xl] |- e => v .

  crl [ExpLR] : D,rho |- e, el => v, vl
            if D,rho |- e => v /\ D,rho |- el => vl .
```

Note how the condition `F(xl)<= e & D' := D` in the rule `FunR` extracts from the set of declarations `D` the declaration corresponding to the function `F`. The resolution of the conditions through matching modulo associativity and commutativity binds variables `xl`, `e`, and `D'`.

The semantic rules for Boolean expressions are represented in the same way. When a semantic rule has a side condition, like the second rule EqR in Figure 3, this is represented also as a condition in the rewrite rule (see below rule `EqR2`).

```
  rl [BCR1] : D,rho |- T => T .
  rl [BCR2] : D,rho |- F => F .

  rl [BVarR] : D,rho |- bx => rho(bx) .

  crl [BOpR] : D,rho |- be bop be' => Ap(bop,bv,bv')
           if D,rho |- be  => bv  /\  D,rho |- be' => bv' .

  crl [NotR1] : D,rho |- Not be => F
            if D,rho |- be => T .
  crl [NotR2] : D,rho |- Not be => T
            if D,rho |- be => F .

  crl [EqR1] : D,rho |- Equal(e,e') => T
           if D,rho |- e  => v  /\  D,rho |- e' => v .
  crl [EqR2] : D,rho |- Equal(e,e') => F
           if D,rho |- e  => v  /\  D,rho |- e' => v' /\ v =/= v' .
endm
```

The module `EVALUATION` is an admissible module, directly executable in Maude 2.0. Next we show some examples. In [36] this semantics is illustrated using as an example the program $\langle Rem(\mathbf{3},\mathbf{5}), D\rangle$, where D is the declaration of function $Rem(x, y)$ that calculates the remainder of dividing y by x. The set of declarations `exDec1` given in the module `FPL-SYNTAX` in Section 3.1 already includes a recursive declaration of the function `Rem`, together with a recursive declaration of the factorial function `Fac`. The next command evaluates the program above in our Maude implementation.

```
Maude> rew exDec1, mt |- FV('Rem)(s(s(s(0))), s(s(s(s(s(0)))))) .
rewrites: 240 in 0ms cpu (3ms real) (~ rewrites/second)
result Num: s(s(0))
```

Maude 2.0 takes approximately 3 milliseconds in rewriting this term, quite simple, and about 1.5 seconds in calculating the factorial of 9:

```
Maude> rew exDec1, mt |- FV('Fac)(s(s(s(s(s(s(s(s(s(0)))))))))) .
rewrites: 409612 in 0ms cpu (1421ms real) (~ rewrites/second)
result Num: 362880
```

where we have adapted the output to use the decimal representation. Most of this time is used in the equations of the operation `Ap`. If we modify the syntax by using the predefined sort `Nat` as a subsort of `Num`, and we use the predefined builtin operations in the definition of `Ap`, the efficiency profit is considerable, as the following examples show:

```
Maude> rew exDec1, mt |- FV('Fac)(9) .
rewrites: 418 in 0ms cpu (5ms real) (~ rewrites/second)
result NzNat: 362880
```

```
Maude> rew exDec1, mt |- FV('Fac)(42) .
rewrites: 1813 in 0ms cpu (27ms real) (~ rewrites/second)
result NzNat: 1405006117752879898543142606244511569936384000000000
```

We can use the `search` command to check that a given expression can only be reduced to a unique value, that is, that the semantics is deterministic.

```
Maude> search exDec1, mt |- FV('Fac)(s(s(s(0)))) =>+ V:Num .
Solution 1 (state 1)
V:Num --> s(s(s(s(s(s(0))))))

No more solutions.
```

This command is also useful to prove that a given transition is possible in the semantics, that is, that it is derivable by using the semantic rules. For example, the following execution proves that the judgement $D, \rho \vdash \mathit{Fac}(2) \Longrightarrow_A 2$ is derivable in the *Fpl* evaluation semantics, where *Fac* is the factorial function.

```
Maude> search exDec1, mt |- FV('Fac)(s(s(0))) =>+ s(s(0)) .
Solution 1 (state 1)
empty substitution

No more solutions.
```

We can also ask Maude to trace the rewriting process, showing us in which order the rules are applied. In order to be able to show here the result, we can only trace a very simple example. The next trace, modified by hand to clarify the steps, shows how the evaluation semantics rules are applied to calculate the factorial of 1. The numbers used to enumerate the rule applications correspond to the numbers used to enumerate the different judgements in the derivation tree in Figure 4. In this tree we do not show the set of function declarations, that does not change throughout the proof.

```
Maude> rew exDec1, mt |- FV('Fac)(s(0)) .

*** rule CR                                                        (1)
exDec1,mt |- s(0)
--->
s(0)

*** rule VarR                                                      (2)
exDec1,V('x) = s(0) |- V('x)
--->
s(0)

*** rule CR                                                        (3)
exDec1,V('x) = s(0) |- 0
--->
0

*** rule EqR2                                                      (4)
exDec1,V('x) = s(0) |- Equal(V('x), 0)
--->
F

*** rule VarR                                                      (5)
exDec1,V('x) = s(0) |- V('x)
--->
s(0)
```

$$\dfrac{mt \vdash 1 \Longrightarrow_A 1^{\,(1)} \qquad \dfrac{\dfrac{\begin{array}{c} x=1 \vdash x \Longrightarrow_A 1^{\,(2)} \\ x=1 \vdash 0 \Longrightarrow_A 0^{\,(3)} \end{array}}{x=1 \vdash \mathsf{Eq}(x,0) \Longrightarrow_B \mathsf{F}^{\,(4)}} \qquad \dfrac{x=1 \vdash x \Longrightarrow_A 1^{\,(5)} \qquad \dfrac{\dfrac{\begin{array}{c} x=1 \vdash x \Longrightarrow_A 1^{\,(6)} \\ x=1 \vdash 1 \Longrightarrow_A 1^{\,(7)} \end{array}}{x=1 \vdash x-1 \Longrightarrow_A 0^{\,(8)}} \qquad \dfrac{\dfrac{\begin{array}{c} x=0 \vdash x \Longrightarrow_A 0^{\,(9)} \\ x=0 \vdash 0 \Longrightarrow_A 0^{\,(10)} \end{array}}{x=0 \vdash \mathsf{Eq}(x,0) \Longrightarrow_B \mathsf{T}^{\,(11)}} \qquad x=0 \vdash 1 \Longrightarrow_A 1^{\,(12)}}{x=0 \vdash \mathsf{If}\ldots \Longrightarrow_A 1^{\,(13)}}}{x=1 \vdash \mathtt{Fac}(x-1) \Longrightarrow_A 1^{\,(14)}}}{x=1 \vdash x * \mathtt{Fac}(x-1) \Longrightarrow_A 1^{\,(15)}}}{x=1 \vdash \mathsf{If}\ \mathsf{Eq}(x,0)\ \mathsf{Then}\ 1\ \mathsf{Else}\ x * \mathtt{Fac}(x-1) \Longrightarrow_A 1^{\,(16)}}}{mt \vdash \mathtt{Fac}(1) \Longrightarrow_A 1^{\,(17)}}$$

Figure 4: Derivation tree for $\mathtt{Fac}(1)$ (call-by-value).

```
*** rule VarR                                                                   (6)
exDec1,V('x) = s(0) |- V('x)
--->
s(0)

*** rule CR                                                                     (7)
exDec1,V('x) = s(0) |- s(0)
--->
s(0)

*** rule OpR                                                                    (8)
exDec1,V('x) = s(0) |- V('x) - s(0)
--->
Ap(-, s(0), s(0)) = 0

*** rule VarR                                                                   (9)
exDec1,V('x) = 0 |- V('x)
--->
0

*** rule CR                                                                    (10)
exDec1,V('x) = 0 |- 0
--->
0

*** rule EqR1                                                                  (11)
exDec1,V('x) = 0 |- Equal(V('x), 0)
--->
T

*** rule CR                                                                    (12)
exDec1,V('x) = 0 |- s(0)
--->
s(0)

*** rule IfR1                                                                  (13)
exDec1,V('x) = 0 |- If Equal(V('x), 0) Then s(0)
                    Else V('x) * FV('Fac)(V('x) - s(0))
--->
s(0)

*** rule FunR                                                                  (14)
exDec1,V('x) = s(0) |- FV('Fac)(V('x) - s(0))
--->
s(0)

*** rule OpR                                                                   (15)
exDec1,V('x) = s(0) |- V('x) * FV('Fac)(V('x) - s(0))
--->
Ap(*, s(0), s(0)) = s(0)

*** rule IfR2                                                                  (16)
exDec1,V('x) = s(0) |- If Equal(V('x), 0) Then s(0)
                       Else V('x) * FV('Fac)(V('x) - s(0))
--->
s(0)

*** rule FunR                                                                  (17)
exDec1,mt |- FV('Fac)(s(0))
--->
s(0)

rewrites: 60 in 0ms cpu (199ms real) (~ rewrites/second)
result Num: s(0)
```

We said above that the rule FunR corresponds to *call-by-value*. The alternative *call-by-name* does not evaluate the parameters and simply substitutes them directly in the body of the definition. The rule describing this behaviour is the following:

$$\text{FunR}' \quad \frac{D, \rho \vdash e[e_1/x_1, \ldots, e_k/x_k] \Longrightarrow_A v}{D, \rho \vdash F(e_1, \ldots, e_k) \Longrightarrow_A v} \quad F(x_1, \ldots, x_k) \Leftarrow e \text{ is in } D$$

where a simultaneous substitution operation is used to substitute the expressions $e_1, \ldots, e_k$ in the actual parameters for variables $x_1, \ldots, x_k$ in an expression e.

The definition of this substitution operation $e[e'/v]$ has to take into account the peculiarities of free and bound variables, to avoid the capture of variables, and in such a way that only free variables are substituted. This substitution may have to introduce new variables that do not appear either in e or in e'. The following functional module `SUBSTITUTION` defines this operation. In the case of simultaneous substitution $e[e_1/x_1, \ldots, e_k/x_k]$, we assume that the substituted variables only occur in e, so it is reduced to several simple substitutions.

The operation `new`, given a (finite) set of variables VS, returns a variable not in VS. To obtain this value variables $z1$, $z2$, etc., are tried until a variable not in the set is found. A new variable is needed when substituting in a let expression that declares a variable also occurring in the substituting expression.

```
fmod SUBSTITUTION is
  protecting FPL-SYNTAX .
  protecting STRING .
  protecting NUMBER-CONVERSION .

  sort VarSet .

  op mt : -> VarSet .
  subsort Var < VarSet .
  op _U_ : VarSet VarSet -> VarSet [assoc comm id: mt] .
  eq x U x = x . *** idempotency

  *** FVar returns the set of free variables in an expression.
  op FVar : Exp -> VarSet .
  op FVar : BExp -> VarSet .
  op FVar : ExpList -> VarSet .

  op _in_ : Var VarSet -> Bool .
  op _not-in_ : Var VarSet -> Bool .
  op _\_ : VarSet VarSet -> VarSet .
  op new : VarSet -> Var .
  op new : VarSet Nat -> Var .
  op newvar : Nat -> Var .

  var n : Num .
  vars x y x' : Var .
  vars e e' e1 e2 : Exp .
  var op : Op .
  var F : FunVar .
  var bx : BVar .
  vars be be1 be2 : BExp .
  var bop : BOp .
  var el : ExpList .
  vars VS VS' : VarSet .
  var N : Nat .
  var xl : VarList .

  eq FVar(n) = mt .
  eq FVar(x) = x .
  eq FVar(e1 op e2) = FVar(e1) U FVar(e2) .
  eq FVar(If be Then e1 Else e2) = FVar(be) U FVar(e1) U FVar(e2) .
  eq FVar(let x = e in e') = (FVar(e') \ x) U FVar(e) .
  eq FVar(F(el)) = FVar(el) .
  eq FVar(e,el) = FVar(e) U FVar(el) .
  eq FVar(T) = mt .
  eq FVar(F) = mt .
  eq FVar(Not be) = FVar(be) .
  eq FVar(be1 bop be2) = FVar(be1) U FVar(be2) .
  eq FVar(bx) = mt .
  eq FVar(Equal(e1,e2)) = FVar(e1) U FVar(e2) .

  eq x in mt = false .
  eq x in (y U VS) = (x == y) or (x in VS) .

  eq x not-in VS = not (x in VS) .

  eq (mt \ VS') = mt .
  eq (y U VS) \ VS' = if (y in VS') then VS \ VS'
                      else y U (VS \ VS') fi .

  eq newvar(N) = V(qid("z" + string(N,10))) .

  eq new(VS) = new(VS, 1) .
  eq new(VS, N) = if newvar(N) not-in VS then newvar(N)
                  else new(VS, N + 1) fi .

  *** substitution of an expression for a variable

  op _[_/_] : Exp Exp Var -> Exp .
  op _[_/_] : BExp Exp Var -> BExp .
  op _[_/_] : ExpList Exp Var -> ExpList .

  eq y [e' / x] = if x == y then e' else y fi .

  eq n [e' / x] = n .

  eq (e1 op e2) [e' / x] = (e1 [e' / x]) op (e2 [e' / x]) .

  eq (If be Then e1 Else e2) [e' / x] =
      If (be[e' / x]) Then (e1[e' / x]) Else (e2[e' / x]) .

  eq (let x = e1 in e2) [e' / x] = let x = (e1 [e' / x]) in e2 .

  ceq (let y = e1 in e2) [e' / x] =
       let y = (e1 [e' / x]) in (e2 [e' / x])
       if x =/= y /\ y not-in FVar(e') .

  ceq (let y = e1 in e2) [e' / x] =
       let x' = (e1 [e' / x]) in ((e2[x' / y]) [e' / x])
       if x =/= y /\ y in FVar(e') /\
          x' := new(FVar(e') U FVar(e2)) .

  eq F(el) [e' / x] = F(el [e' / x]) .

  eq (e, el) [e' / x] =  (e[e' / x]), (el[e' / x]) .

  eq T [e' / x] = T .
  eq F [e' / x] = F .

  eq bx [e' / x] = bx .

  eq (be1 bop be2) [e' / x] = (be1 [e' / x]) bop (be2 [e' / x]) .
```

```
  eq (Not be) [e' / x] = Not (be[e' / x]) .

  eq Equal(e1,e2) [e' / x] = Equal(e1[e' / x],e2[e' / x]) .

  *** multiple simultaneous substitution

  op _[_/_] : Exp ExpList VarList -> Exp .

  eq e [e', el / x, xl] = (e [e' / x])[el / xl] .

endfm
```

Once the substitution is defined, we can write the rewrite rule that implements call-by-name:

```
*** call-by-name
  crl [FunR'] : D,rho |- F(el) => v
            if F(xl)<= e & D' := D  /\  D,rho |- (e[el / xl]) => v .
```

We can test this semantics with the program $\langle Rem(\mathbf{3},\mathbf{5}), D\rangle$, evaluated by the next command:

```
Maude> rew exDec1, mt |- FV('Rem)(s(s(s(0))), s(s(s(s(s(0)))))) .
rewrites: 234 in 0ms cpu (1ms real) (~ rewrites/second)
result Num: s(s(0))
```

We can also trace the evaluation of $Fac(1)$, to check how the computation of an expression is affected by this change. The corresponding derivation tree is shown in Figure 5.

```
Maude> rew exDec1, mt |- FV('Fac)(s(0)) .

*** rule CR                                                          (1)
exDec1,mt |- s(0)
--->
s(0)

*** rule CR                                                          (2)
exDec1,mt |- 0
--->
0

*** rule EqR2                                                        (3)
exDec1,mt |- Equal(s(0), 0)
--->
F

*** rule CR                                                          (4)
exDec1,mt |- s(0)
--->
s(0)

*** rule CR                                                          (5)
exDec1,mt |- s(0)
--->
s(0)

*** rule CR                                                          (6)
exDec1,mt |- s(0)
--->
s(0)

*** rule OpR                                                         (7)
exDec1,mt |- s(0) - s(0)
--->
Ap(-, s(0), s(0)) = 0
```

$$
\dfrac{
\dfrac{
\dfrac{\begin{array}{c} mt \vdash 1 \Longrightarrow_A 1 \;{\scriptstyle (1)} \\ mt \vdash 0 \Longrightarrow_A 0 \;{\scriptstyle (2)} \end{array}}{mt \vdash \mathsf{Eq}(1,0) \Longrightarrow_B \mathsf{F} \;{\scriptstyle (3)}}
\quad
\dfrac{mt \vdash 1 \Longrightarrow_A 1 \;{\scriptstyle (4)} \quad
\dfrac{
\dfrac{
\dfrac{
\dfrac{\begin{array}{c} mt \vdash 1 \Longrightarrow_A 1 \;{\scriptstyle (5)} \\ mt \vdash 1 \Longrightarrow_A 1 \;{\scriptstyle (6)} \end{array}}{mt \vdash 1-1 \Longrightarrow_A 0 \;{\scriptstyle (7)}}
\quad mt \vdash 0 \Longrightarrow_A 0 \;{\scriptstyle (8)}
}{mt \vdash \mathsf{Eq}(1-1,0) \Longrightarrow_B \mathsf{T} \;{\scriptstyle (9)}}
\quad mt \vdash 1 \Longrightarrow_A 1 \;{\scriptstyle (10)}
}{mt \vdash \mathsf{If}\ \mathsf{Eq}(1-1,0)\ \mathsf{Then}\ 1\ \mathsf{Else}\ (1-1) * \mathtt{Fac}((1-1)-1) \Longrightarrow_A 1 \;{\scriptstyle (11)}}
}{mt \vdash \mathtt{Fac}(1-1) \Longrightarrow_A 1 \;{\scriptstyle (12)}}
}{mt \vdash 1 * \mathtt{Fac}(1-1) \Longrightarrow_A 1 \;{\scriptstyle (13)}}
}{mt \vdash \mathsf{If}\ \mathsf{Eq}(1,0)\ \mathsf{Then}\ 1\ \mathsf{Else}\ 1 * \mathtt{Fac}(1-1) \Longrightarrow_A 1 \;{\scriptstyle (14)}}
}{mt \vdash \mathtt{Fac}(1) \Longrightarrow_A 1 \;{\scriptstyle (15)}}
$$

Figure 5: Derivation tree for `Fac`(1) (call-by-name).

```
*** rule CR                                                             (8)
exDec1,mt |- 0
--->
0

*** rule EqR1                                                           (9)
exDec1,mt |- Equal(s(0) - s(0), 0)
--->
T

*** rule CR                                                            (10)
exDec1,mt |- s(0)
--->
s(0)

*** rule IfR1                                                          (11)
exDec1,mt |- If Equal(s(0) - s(0), 0) Then s(0) Else
             (s(0) - s(0)) * FV('Fac)((s(0) - s(0)) - s(0))
--->
s(0)

*** rule FunR'                                                         (12)
exDec1,mt |- FV('Fac)(s(0) - s(0))
--->
s(0)

*** rule OpR                                                           (13)
exDec1,mt |- s(0) * FV('Fac)(s(0) - s(0))
--->
Ap(*, s(0), s(0)) = s(0)

*** rule IfR2                                                          (14)
exDec1,mt |- If Equal(s(0), 0) Then s(0)
              Else s(0) * FV('Fac)(s(0) - s(0))
--->
s(0)

*** rule FunR'                                                         (15)
exDec1,mt |- FV('Fac)(s(0))
--->
s(0)

rewrites: 65 in 0ms cpu (158ms real) (~ rewrites/second)
result Num: s(0)
```

3.3 Computation semantics

In this section we implement a computation (or small step) semantics for the language *Fpl* that describes the sequence of primitive operations that the evaluation of an expression gives rise to. As in the evaluation semantics, variable environments and function declarations are needed. The semantic judgements to evaluate arithmetic and Boolean expressions are $D, \rho \vdash e \longrightarrow_A e'$ and $D, \rho \vdash be \longrightarrow_B be'$. The semantic rules that define these judgements for $\longrightarrow_A$ and $\longrightarrow_B$ are shown in Figure 6 and 7, respectively.

In the implementation of this semantics we use the module FPL-SYNTAX given in Section 3.1, although with some modifications. For expressing with easiness and clarity the rule FunRc in Figure 6, we need that the operation for building lists of expressions has as identity element the empty list, as we shall see:

```
  subsort Exp < ExpList .
  op nil : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: nil prec 30] .
```

VarRc $$\frac{}{D, \rho \vdash x \longrightarrow_A \rho(x)}$$

OpRc $$\frac{}{D, \rho \vdash v \ op \ v' \longrightarrow_A Ap(op, v, v')}$$

$$\frac{D, \rho \vdash e \longrightarrow_A e''}{D, \rho \vdash e \ op \ e' \longrightarrow_A e'' \ op \ e'} \qquad \frac{D, \rho \vdash e' \longrightarrow_A e''}{D, \rho \vdash e \ op \ e' \longrightarrow_A e \ op \ e''}$$

IfRc $$\frac{D, \rho \vdash be \longrightarrow_B be'}{D, \rho \vdash \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e' \longrightarrow_A \mathsf{If}\ be'\ \mathsf{Then}\ e\ \mathsf{Else}\ e'}$$

$$\frac{}{D, \rho \vdash \mathsf{If\ T\ Then}\ e\ \mathsf{Else}\ e' \longrightarrow_A e} \qquad \frac{}{D, \rho \vdash \mathsf{If\ F\ Then}\ e\ \mathsf{Else}\ e' \longrightarrow_A e'}$$

LocRc $$\frac{D, \rho \vdash e \longrightarrow_A e''}{D, \rho \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \longrightarrow_A \mathsf{let}\ x = e''\ \mathsf{in}\ e'} \qquad \frac{}{D, \rho \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e' \longrightarrow_A e'[v/x]}$$

FunRc $$\frac{D, \rho \vdash e_i \longrightarrow_A e_i'}{D, \rho \vdash F(e_1, \ldots, e_i, \ldots, e_k) \longrightarrow_A F(e_1, \ldots, e_i', \ldots, e_k)}$$

$$\frac{}{D, \rho \vdash F(v_1, \ldots, v_k) \longrightarrow_A e[v_1/x_1, \ldots, v_k/x_k]}\ F(x_1, \ldots, x_k) \Leftarrow e \text{ is in } D$$

Figure 6: Computation semantics for *Fpl*, $\longrightarrow_A$.

BVarRc $$\frac{}{D, \rho \vdash bx \longrightarrow_B \rho(bx)}$$

BOpRc $$\frac{}{D, \rho \vdash bv \ bop \ bv' \longrightarrow_B Ap(bop, bv, bv')}$$

$$\frac{D, \rho \vdash be \longrightarrow_B be''}{D, \rho \vdash be \ bop \ be' \longrightarrow_B be'' \ bop \ be'} \qquad \frac{D, \rho \vdash be' \longrightarrow_B be''}{D, \rho \vdash be \ bop \ be' \longrightarrow_B be \ bop \ be''}$$

NotRc $$\frac{D, \rho \vdash be \longrightarrow_B be'}{D, \rho \vdash \mathsf{Not}\ be \longrightarrow_B \mathsf{Not}\ be'} \qquad \frac{}{D, \rho \vdash \mathsf{Not\ T} \longrightarrow_B \mathsf{F}} \qquad \frac{}{D, \rho \vdash \mathsf{Not\ F} \longrightarrow_B \mathsf{T}}$$

EqRc $$\frac{D, \rho \vdash e \longrightarrow_A e''}{D, \rho \vdash \mathsf{Equal}(e, e') \longrightarrow_B \mathsf{Equal}(e'', e')} \qquad \frac{D, \rho \vdash e' \longrightarrow_A e''}{D, \rho \vdash \mathsf{Equal}(e, e') \longrightarrow_B \mathsf{Equal}(e', e'')}$$

$$\frac{v = v'}{D, \rho \vdash \mathsf{Equal}(v, v') \longrightarrow_B \mathsf{T}} \qquad \frac{v \neq v'}{D, \rho \vdash \mathsf{Equal}(v, v') \longrightarrow_B \mathsf{F}}$$

Figure 7: Computation semantics for Boolean expressions, $\longrightarrow_B$.

The modules `AP`, `ENV`, and `SUBSTITUTION` are used without modification. The following module `COMPUTATION` contains the implementation of the new semantic rules. The same techniques as in the previous section are used. Note how the rule `FunRc1` expresses the non-deterministic choice of one of the arguments to be rewritten, by means of the pattern matching between the list of arguments and the pattern `el,e,el'` modulo associativity and identity (the empty list). This pattern also includes the cases with one or two arguments, which are handled by making empty some of the lists.

```
mod COMPUTATION is
  protecting AP .  protecting ENV .  protecting SUBSTITUTION .

  sort Statement .
  subsorts Num Boolean < Statement .

  op _,_|-_ : Dec ENV Exp -> Statement [prec 40] .
  op _,_|-_ : Dec ENV BExp -> Statement [prec 40] .

  vars D D' : Dec .   var rho : ENV .     vars e e' e'' : Exp .
  var bx : BVar .     vars v v' : Num .   vars bv bv' : Boolean .
  var op : Op .       var x : Var .       vars be be' be'' : BExp .
  var xl : VarList .  var vl : NumList .  vars el el' : ExpList .
  var bop : BOp .     var F : FunVar .

  *** computation semantics for Fpl

  rl  [VarRc] : D,rho |- x => rho(x) .

  rl  [OpRc1] : D,rho |- v op v' => Ap(op,v,v') .

  crl [OpRc2] : D,rho |- e op e' => e'' op e'
           if D,rho |- e  => e'' .
  crl [OpRc3] : D,rho |- e op e' => e op e''
           if D,rho |- e'  => e'' .

  crl [IfRc1] : D,rho |- If be Then e Else e' => If be' Then e Else e'
           if D,rho |- be => be' .
  rl  [IfRc2] : D,rho |- If T Then e Else e' => e .
  rl  [IfRc3] : D,rho |- If F Then e Else e' => e' .

  crl [LocRc1] : D,rho |- let x = e in e' => let x = e'' in e'
            if D,rho |- e  => e'' .
  rl  [LocRc2] : D,rho |- let x = v in e' => e'[v / x] .

  crl [FunRc1] : D,rho |- F(el,e,el') => F(el,e',el')
            if D,rho |- e => e' .
  crl [FunRc2] : D,rho |- F(vl) => e[vl / xl]
            if F(xl)<= e & D' := D .

  *** computation semantics for boolean expressions

  rl [BVarRc] : D,rho |- bx => rho(bx) .

  rl  [BOpRc1] : D,rho |- bv bop bv' => Ap(bop,bv,bv') .
  crl [BOpRc2] : D,rho |- be bop be' => be'' bop be'
            if D,rho |- be  => be'' .
  crl [BOpRc3] : D,rho |- be bop be' => be bop be''
            if D,rho |- be'  => be'' .

  crl [NotRc1] : D,rho |- Not be => Not be'
            if D,rho |- be => be' .
  rl  [NotRc2] : D,rho |- Not T => F .
  rl  [NotRc3] : D,rho |- Not F => T .
  crl [EqRc1] : D,rho |- Equal(e,e') => Equal(e'',e')
            if D,rho |- e => e'' .
  crl [EqRc2] : D,rho |- Equal(e,e') => Equal(e,e'')
            if D,rho |- e' => e'' .
  crl [EqRc3] : D,rho |- Equal(v,v') => T
            if v == v' .
  crl [EqRc4] : D,rho |- Equal(v,v') => F
            if v =/= v' .
endm
```

We can use this implementation of the computation semantics for evaluating the expression $Rem(\mathbf{3}, \mathbf{5})$, considered in the previous section.

```
Maude> rew exDec1, mt |- FV('Rem)(s(s(s(0))), s(s(s(s(s(0)))))) .
rewrites: 65 in 0ms cpu (1ms real) (~ rewrites/second)
result Exp: If Equal(s(s(s(0))), s(s(s(s(s(0)))))) Then 0
            Else If Equal(s(s(s(s(s(0))))) - s(s(s(0))), 0) Then s(s(s(s(s(0)))))
                 Else FV('Rem)(s(s(s(0))),s(s(s(s(s(0))))) - s(s(s(0))))
```

What we have obtained is the resulting expression after the *first* step. Although the module `COMPUTATION` is admissible, we cannot use it directly, in an easy way, to know to which final value an expression is evaluated. The reason is that, since we are working now with a computation semantics, each rule (or rewrite) represents *one* step. The righthand sides of the rewrite rules are expressions, although the lefthand sides are terms of sort `Statement` with a set of declarations, an environment, and an expression. In this way, the application of these rules cannot be concatenated by using the transitivity rule of rewriting logic, since once we apply a rule the resulting term does no longer match the lefthand side of any rule. (On the other hand, this structure of their lefthand side ensures that the rules are only applied at the top, thus avoiding undesired rewrite steps inside expressions, for example.)

We can solve this problem by implementing the reflexive, transitive closure of transitions $\longrightarrow_A$ and $\longrightarrow_B$. Consider the first one; if we implemented it as follows:

```
  rl  [zero] : D,rho |- v => v .  *** no step
  crl [more] : D,rho |- e => v
           if D,rho |- e => e'   *** one step
           /\ D,rho |- e' => v . *** all the rest
```

then we would have executability problems, since the rules `zero` and `more` themselves could be used to try to resolve the first condition of rule `more`, giving rise to infinite loops.

To avoid this problem, we use different constructors to build the terms in the lefthand side of these rules. In this way, we control which rules can be applied to resolve each one of the conditions.

```
  op _,_|=_ : Dec ENV Exp -> Statement [prec 40] .
  op _,_|=_ : Dec ENV BExp -> Statement [prec 40] .

  rl  [zero] : D,rho |= v => v .
  crl [more] : D,rho |= e => v
           if D,rho |- e => e'  /\  D,rho |= e' => v .

  rl  [zero] : D,rho |= bv => bv .
  crl [more] : D,rho |= be => bv
           if D,rho |- be => be'  /\  D,rho |= be' => bv .
```

Now we can use the complete implementation for evaluating the expression $Rem(\mathbf{3}, \mathbf{5})$.

```
Maude> rew exDec1, mt |= FV('Rem)(s(s(s(0))), s(s(s(s(s(0)))))) .
rewrites: 181 in 0ms cpu (1ms real) (~ rewrites/second)
result Num: s(s(0))
```

We can also show the trace produced when $Fac(1)$ is evaluated. To simplify we only show transitions of the relations $\longrightarrow_A$ and $\longrightarrow_B$, and we have removed the applications of rule `zero` (once) and rule `more` (8 times), at the end of the trace.

```
Maude> rew exDec1, mt |= FV('Fac)(s(0)) .

*** rule FunRc2
exDec1,mt |- FV('Fac)(s(0))
--->
If Equal(s(0), 0) Then s(0) Else s(0) * FV('Fac)(s(0)) - s(0))

*** rule EqRc4
exDec1,mt |- Equal(s(0), 0)
--->
F

*** rule IfRc1
exDec1,mt |- If Equal(s(0), 0) Then s(0) Else s(0) * FV('Fac)(s(0) - s(0))
--->
If F Then s(0) Else s(0) * FV('Fac)(s(0) - s(0))

*** rule IfRc3
exDec1,mt |- If F Then s(0) Else s(0) * FV('Fac)(s(0) - s(0))
--->
s(0) * FV('Fac)(s(0) - s(0))

*** rule OpRc1
exDec1,mt |- s(0) - s(0)
--->
Ap(-, s(0), s(0)) = 0

*** rule FunRc1
exDec1,mt |- FV('Fac)(s(0) - s(0))
--->
FV('Fac)(nil,0,nil)

*** rule OpRc3
exDec1,mt |- s(0) * FV('Fac)(s(0) - s(0))
--->
s(0) * FV('Fac)(0)

*** rule FunRc2
exDec1,mt |- FV('Fac)(0)
--->
If Equal(0, 0) Then s(0) Else 0 * FV('Fac)(0 - s(0))

*** rule OpRc3
exDec1,mt |- s(0) * FV('Fac)(0)
--->
s(0) * (If Equal(0, 0) Then s(0) Else 0 * FV('Fac)(0 - s(0)))

*** rule EqRc3
exDec1,mt |- Equal(0, 0)
--->
T

*** rule IfRc1
exDec1,mt |- If Equal(0, 0) Then s(0) Else 0 * FV('Fac)(0 - s(0))
--->
If T Then s(0) Else (0).Num * FV('Fac)(0 - s(0))

*** rule OpRc3
exDec1,mt |- s(0) * (If Equal(0, 0) Then s(0) Else 0 * FV('Fac)(0 - s(0)))
```

$$\text{LocRc} \quad \frac{D,\rho \vdash e \longrightarrow_A e''}{D,\rho \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \longrightarrow_A \mathsf{let}\ x = e''\ \mathsf{in}\ e'}$$

$$\frac{D,\rho[v/x] \vdash e \longrightarrow_A e'}{D,\rho \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow_A \mathsf{let}\ x = v\ \mathsf{in}\ e'} \qquad \frac{}{D,\rho \vdash \mathsf{let}\ x = v\ \mathsf{in}\ v' \longrightarrow_A v'}$$

$$\text{FunRc} \quad \frac{D,\rho \vdash e_i \longrightarrow_A e'_i}{D,\rho \vdash F(e_1,\ldots,e_i,\ldots,e_k) \longrightarrow_A F(e_1,\ldots,e'_i,\ldots,e_k)}$$

$$\frac{}{D,\rho \vdash F(v_1,\ldots,v_k) \longrightarrow_A \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ x_k = v_k\ \mathsf{in}\ e}\ F(x_1,\ldots,x_k) \Leftarrow e \text{ is in } D$$

Figure 8: Modification of the rules without using substitutions.

```
--->
s(0) * (If T Then s(0) Else 0 * FV('Fac)(0 - s(0)))

*** rule IfRc2
exDec1,mt |- If T Then s(0) Else 0 * FV('Fac)(0 - s(0))
--->
s(0)

*** rule OpRc3
exDec1,mt |- s(0) * (If T Then s(0) Else 0 * FV('Fac)(0 - s(0)))
--->
s(0) * s(0)

*** rule OpRc1
exDec1,mt |- s(0) * s(0)
--->
Ap(*, s(0), s(0)) = s(0)

rewrites: 68 in 0ms cpu (255ms real) (~ rewrites/second)
result Num: s(0)
```

The semantic rules in Figure 6 use the syntactic substitution of values for variables in an expression. However, environments were precisely introduced with the purpose of keeping the bindings between variables and values, so it could be preferable not hiding part of this goal by using substitutions. One way of removing the use of substitutions is to define the semantic rules LocRc and FunRc as it is done in Figure 8.

The following rewrite rules implement these new semantic rules. We have used an auxiliary operation `buildLet` to build the result expression in the second rule FunRc. Specifically, the operation `buildLet` takes as arguments a list $x_1,\ldots,x_n$ of variables, a list $v_1,\ldots,v_n$ of values such that v_i is the value to which variable x_i has to be bound, and an expression e, and it returns the expression $\mathsf{let}\ x_1 = v_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ x_k = v_k\ \mathsf{in}\ e$. The following equations define it recursively on the received lists.

```
  op buildLet : VarList NumList Exp -> Exp .
  eq buildLet(nil, nil, e) = e .
  eq buildLet(x, v, e) = let x = v in e .
  eq buildLet((x, xl), (v, vl), e) = let x = v in buildLet(xl, vl, e) .

  crl [LocRc1] : D,rho |- let x = e in e' => let x = e'' in e'
              if D,rho |- e  => e'' .
  crl [LocRc2] : D,rho |- let x = v in e => let x = v in e'
              if D,rho[v / x] |-  e => e' .
  rl  [LocRc3] : D,rho |- let x = v in v' => v' .

  crl [FunRc1] : D,rho |- F(el,e,el') => F(el,e',el')
```

1. Syntactic categories

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
st & \in & \mathit{States} & C & \in & \mathit{Control} & v & \in & \mathit{Num} \\
S & \in & \mathit{Stack} & env & \in & \mathit{Env} & bv & \in & \mathit{Boolean} \\
a & \in & \mathit{Assoc} & cons & \in & \mathit{Constants} & F & \in & \mathit{FunVar}
\end{array}$$

2. Definitions

$$\begin{array}{rcl}
st & ::= & \langle S, env, C\rangle \\
S & ::= & \varepsilon \mid v.S \mid bv.S \\
C & ::= & \varepsilon \mid e.C \mid cons.C \\
cons & ::= & op \mid bop \mid \langle x, e\rangle \mid \mathsf{if}(e, e') \mid \mathsf{not} \mid \mathsf{equal} \mid F \mid \mathsf{pop} \\
env & ::= & \varepsilon \mid a.env \\
a & ::= & (x, v) \mid (bx, bv) \mid (F, (x, e)) \\
v & ::= & \mathbf{n} \\
bv & ::= & \mathsf{T} \mid \mathsf{F}
\end{array}$$

Figure 9: States of the abstract machine for *Fpl*.

```
            if D,rho |- e => e' .
  crl [FunRc2] : D,rho |- F(vl) => buildLet(xl, vl, e)
            if F(xl)<= e & D' := D .
```

We can evaluate again the expression $\mathit{Rem}(\mathbf{3}, \mathbf{5})$ with the new semantics. The result obviously coincides with the one obtained previously.

```
Maude> rew exDec1, mt |= FV('Rem)(s(s(s(0))), s(s(s(s(s(0)))))) .
rewrites: 527 in 0ms cpu (9ms real) (~ rewrites/second)
result Num: s(s(0))
```

3.3.1 Abstract machine for *Fpl*

In this section we present a concrete operational semantics for *Fpl* based on an abstract machine. We obtain in this way a formal interpreter of the language that is executed in a virtual machine. The states of this machine are tuples $\langle S, env, C\rangle$, where S is a stack of values, *env* is an environment (a finite list of bindings between variables and values, in this case), and C is a control sequence. The abstract syntax of these states is shown in Figure 9.

To describe how the machine works, it is enough to define a relation $\longrightarrow$ between states,

$$\langle S, env, C\rangle \longrightarrow \langle S', env', C'\rangle.$$

The semantic rules that define this relation are shown in Figures 10 and 11.

The rules Varm and Funm2 use in their premises predicates like $env, x \vdash v$ to point out that v is the value associated to variable x in the environment *env*. We do not show the obvious definition of these predicates, that will be implemented by means of operations `lookup` in the module `ABS-MACHINE-SEMANTICS` below.

The following functional module `ABS-MACHINE-SYNTAX` defines the syntax of the states of the abstract machine.

```
fmod ABS-MACHINE-SYNTAX is
  protecting AP .

  *** states of the abstract machine for Fpl

  sorts Constants Assoc Env Stack Control States .

  op <_,_,_> : Stack Env Control -> States [prec 60] .

  op mtS : -> Stack .
```

Opm1 $\langle S, env, e\ op\ e'.C\rangle \longrightarrow \langle S, env, e.e'.op.C\rangle$

$\langle S, env, be\ bop\ be'.C\rangle \longrightarrow \langle S, env, be.be'.bop.C\rangle$

Ifm1 $\langle S, env, \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e'.C\rangle \longrightarrow \langle S, env, be.\mathsf{if}(e, e').C\rangle$

Locm1 $\langle S, env, \mathsf{let}\ x = e\ \mathsf{in}\ e'.C\rangle \longrightarrow \langle S, env, e.\langle x, e'\rangle.C\rangle$

Funm1 $\langle S, env, F(e).C\rangle \longrightarrow \langle S, env, e.F.C\rangle$

Notm1 $\langle S, env, \mathsf{Not}\ be.C\rangle \longrightarrow \langle S, env, be.\mathsf{not}.C\rangle$

Eqm1 $\langle S, env, \mathsf{Equal}(e, e').C\rangle \longrightarrow \langle S, env, e.e'.\mathsf{equal}.C\rangle$

Figure 10: Analysis rules for the abstract machine.

Opm2 $\langle v'.v.S, env, op.C\rangle \longrightarrow \langle Ap(op, v, v').S, env, C\rangle$

$\langle bv'.bv.S, env, bop.C\rangle \longrightarrow \langle Ap(bop, bv, bv').S, env, C\rangle$

Varm $\dfrac{env, x \vdash v}{\langle S, env, x.C\rangle \longrightarrow \langle v.S, env, C\rangle}$ $\dfrac{env, bx \vdash bv}{\langle S, env, bx.C\rangle \longrightarrow \langle bv.S, env, C\rangle}$

Valm $\langle S, env, v.C\rangle \longrightarrow \langle v.S, env, C\rangle$ $\langle S, env, bv.C\rangle \longrightarrow \langle bv.S, env, C\rangle$

Notm2 $\langle \mathsf{T}.S, env, \mathsf{not}.C\rangle \longrightarrow \langle \mathsf{F}.S, env, C\rangle$ $\langle \mathsf{F}.S, env, \mathsf{not}.C\rangle \longrightarrow \langle \mathsf{T}.S, env, C\rangle$

Eqm2 $\dfrac{v = v'}{\langle v.v'.S, env, \mathsf{equal}.C\rangle \longrightarrow \langle \mathsf{T}.S, env, C\rangle}$ $\dfrac{v \neq v'}{\langle v.v'.S, env, \mathsf{equal}.C\rangle \longrightarrow \langle \mathsf{F}.S, env, C\rangle}$

Ifm2 $\langle \mathsf{T}.S, env, \mathsf{if}(e, e').C\rangle \longrightarrow \langle S, env, e.C\rangle$ $\langle \mathsf{F}.S, env, \mathsf{if}(e, e').C\rangle \longrightarrow \langle S, env, e'.C\rangle$

Funm2 $\dfrac{env, F \vdash (x, e)}{\langle v.S, env, F.C\rangle \longrightarrow \langle S, (x, v).env, e.\mathsf{pop}.C\rangle}$

Locm2 $\langle v.S, env, \langle x, e\rangle.C\rangle \longrightarrow \langle S, (x, v).env, e.\mathsf{pop}.C\rangle$

Pop $\langle S, (x, v).env, \mathsf{pop}.C\rangle \longrightarrow \langle S, env, C\rangle$

Figure 11: Application rules for the abstract machine.

```
  op _._ : Num Stack -> Stack .
  op _._ : Boolean Stack -> Stack .

  op mtC : -> Control .
  op _._ : Exp Control -> Control [prec 50] .
  op _._ : BExp Control -> Control [prec 50] .
  op _._ : Constants Control -> Control [prec 50] .

  subsort Op BOp < Constants .

  op <_,_> : Var Exp -> Constants .
  op if : Exp Exp -> Constants .
  op not : -> Constants .
  op equal : -> Constants .
  op pop : -> Constants .
  subsort FunVar < Constants .

  op mtE : -> Env .
  subsort Assoc < Env .
  op _._ : Env Env -> Env [assoc id: mtE] .

  op `(_,_`) : Var Num -> Assoc .
  op `(_,_`) : BVar Boolean -> Assoc .
  op `(_,_,_`) : FunVar Var Exp -> Assoc .
endfm
```

The system module ABS-MACHINE-SEMANTICS implements the analysis rules and application rules of the abstract machine. Note that the rules in this module *do not* use rewrites in the conditions, since the semantic rules in Figures 10 and 11 do not have premises with transitions.

```
mod ABS-MACHINE-SEMANTICS is
  protecting ABS-MACHINE-SYNTAX .

  op lookup : Env Var -> Num .
  op lookup : Env BVar -> Boolean .
  op lookup : Env FunVar -> Constants .

  vars x x' : Var .
  vars v v' : Num .
  vars bx bx' : BVar .
  vars bv bv' : Boolean .
  vars FV FV' : FunVar .
  vars e e' : Exp .
  var op : Op .
  var bop : BOp .
  var be be' : BExp .
  var env : Env .
  var S : Stack .
  var C : Control .

  eq lookup((x',v) . env, x) = if x == x' then v else lookup(env,x) fi .
  eq lookup((bx, bv) . env, x) = lookup(env, x) .
  eq lookup((FV,x',e) . env, x) = lookup(env, x) .

  eq lookup((x,v) . env, bx) = lookup(env,bx) .
  eq lookup((bx',bv) . env, bx) = if bx == bx' then bv else lookup(env,bx) fi .
  eq lookup((FV,x',e) . env, bx) = lookup(env,bx) .

  eq lookup((x,v) . env, FV) = lookup(env,FV) .
  eq lookup((bx',bv) . env, FV) = lookup(env,FV) .
  eq lookup((FV',x',e) . env, FV) =
     if FV == FV' then < x', e > else lookup(env,FV) fi .
  *** Analysis rules for the abstract machine

  rl [Opm1]  : < S, env, e op e' . C > => < S, env, (e . e' . op . C) > .
  rl [Opm1'] : < S, env, be bop be' . C > => < S, env, be . be' . bop . C > .

  rl [Notm1] : < S, env, Not be . C > => < S, env, be . not . C > .

  rl [Eqm1] : < S, env, Equal(e,e') . C > => < S, env, e . e' . equal . C > .

  rl [Ifm1] : < S, env, If be Then e Else e' . C >
         => < S, env, be . if(e,e') . C > .

  rl [Funm1] : < S, env, FV(e) . C > => < S, env, e . FV . C > .

  rl [Locm1] : < S, env, let x = e in e' . C > => < S, env, e . < x, e' > . C > .


  *** Application rules for the abstract machine

  rl [Opm2]  : < v' . v . S, env, op . C > => < Ap(op, v, v') . S, env, C > .
  rl [Opm2'] : < bv' . bv . S, env, bop . C >
          => < Ap(bop, bv, bv') . S, env, C > .

  crl [Varm]  : < S, env, x . C > => < v . S, env, C >
              if v := lookup(env, x) .
  crl [Varm'] : < S, env, bx . C > => < bv . S, env, C >
              if bv := lookup(env, bx) .

  rl [Valm]  : < S, env, v . C > => < v . S, env, C > .
  rl [Valm'] : < S, env, bv . C > => < bv . S, env, C > .

  rl [Notm2]  : < T . S, env, not . C > => < F . S, env, C > .
  rl [Notm2'] : < F . S, env, not . C > => < T . S, env, C > .

  crl [Eqm2]  : < v . v' . S, env, equal . C > => < T . S, env, C > if v == v' .
  crl [Eqm2'] : < v . v' . S, env, equal . C > => < F . S, env, C > if v =/= v' .

  rl [Ifm2 ] : < T . S, env, if(e,e') . C > => < S, env, e . C > .
  rl [Ifm2'] : < F . S, env, if(e,e') . C > => < S, env, e' . C > .

  crl [Funm2] : < v . S, env, FV . C > => < S, (x, v) . env, e . pop . C >
   if < x, e > := lookup(env,FV) .

  rl [Locm2] : < v . S, env, < x, e > . C > => < S, (x,v) . env, e . pop . C > .

  rl [Pop] : < S, (x,v) . env, pop . C > => < S, env, C > .

endm
```

By calculating the factorial of 1 we can see how the abstract machine works. The rewrites in this semantics are completely deterministic, since there is no rewrite in conditions, there are not two rules with the same lefthand side, and all rewrites are done at the top level. That is the reason why we have modified the trace produced by Maude, in order to show this sequentiality. Moreover, we have written `dec1` instead of the complete declaration of function `Fac`, that we show in the command `rew` introduced to Maude.

```
Maude> rew < mtS, (FV('Fac), V('x), If Equal(V('x),0) Then s(0)
                                    Else V('x) * FV('Fac)(V('x) - s(0))),
             FV('Fac)(s(0)) . mtC > .

< mtS, dec1, FV('Fac)(s(0)) . mtC >
```

```
--->  *** rule Funm1
< mtS, dec1, s(0) . FV('Fac) . mtC >
--->  *** rule Valm
< s(0) . mtS, dec1, FV('Fac) . mtC >
--->  *** rule Funm2
< mtS, (V('x),s(0)) . dec1, If Equal(V('x), 0) Then s(0)
                            Else V('x) * FV('Fac)(V('x) - s(0)) . pop . mtC >
--->  *** rule Ifm1
< mtS, (V('x),s(0)) . dec1, Equal(V('x), 0) . if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . mtC >
--->  *** rule Eqm1
< mtS, (V('x),s(0)) . dec1, V('x) . 0 . equal . if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . mtC >
--->  *** rule Varm
< s(0) . mtS, (V('x),s(0)) . dec1, 0 . equal . if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . mtC >
--->  *** rule Valm
< 0 . s(0) . mtS, (V('x),s(0)) . dec1, equal . if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . mtC >
--->  *** rule Eqm2'
< F . mtS, (V('x),s(0)) . dec1, if(s(0), V('x) * FV('Fac)(V('x) - s(0))) .
    pop . mtC >
--->  *** rule Ifm2'
< mtS, (V('x),s(0)) . dec1, V('x) * FV('Fac)(V('x) - s(0)) . pop . mtC >
--->  *** rule Opm1
< mtS, (V('x),s(0)) . dec1, V('x) . FV('Fac)(V('x) - s(0)) . * . pop . mtC >
--->  *** rule Varm
< s(0) . mtS, (V('x),s(0)) . dec1, FV('Fac)(V('x) - s(0)) . * . pop . mtC >
--->  *** rule Funm1
< s(0) . mtS, (V('x),s(0)) . dec1, V('x) - s(0) . FV('Fac) . * . pop . mtC >
--->  *** rule Opm1
< s(0) . mtS,(V('x),s(0)) . dec1, V('x) . s(0) . - . FV('Fac) . * . pop . mtC >
--->  *** rule Varm
< s(0) . s(0) . mtS, (V('x),s(0)) . dec1, s(0) . - . FV('Fac) . * . pop . mtC >
--->  *** rule Valm
< s(0) . s(0) . s(0) . mtS, (V('x),s(0)) . dec1, - . FV('Fac) . * . pop . mtC >
--->  *** rule Opm2
< 0 . s(0) . mtS, (V('x),s(0)) . dec1, FV('Fac) . * . pop . mtC >
--->  *** rule Funm2
< s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, If Equal(V('x), 0) Then
    s(0) Else V('x) * FV('Fac)(V('x) - s(0)) . pop . * . pop . mtC >
--->  *** rule Ifm1
< s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, Equal(V('x), 0) .
    if(s(0), V('x) * FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Eqm1
< s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, V('x) . 0 . equal .
    if(s(0), V('x) * FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Varm
< 0 . s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, 0 .
    equal . if(s(0), V('x) * FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Valm
< 0 . 0 . s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, equal .
    if(s(0), V('x) * FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Eqm2
< T . s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Ifm2
< s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, s(0) . pop . * . pop . mtC >
--->  *** rule Valm
< s(0) . s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, pop . * . pop . mtC >
--->  *** rule Pop
< s(0) . s(0) . mtS, (V('x),s(0)) . dec1, * . pop . mtC >
--->  *** rule Opm2
```

1. Syntactic categories

$$\begin{array}{lclclclclcl} p & \in & \mathit{Prog} & \quad & C & \in & \mathit{Com} & \quad & x & \in & \mathit{Var} \\ op & \in & \mathit{Op} & & bop & \in & \mathit{BOp} & & bx & \in & \mathit{BVar} \\ e & \in & \mathit{Exp} & & be & \in & \mathit{BExp} & & n & \in & \mathit{Num} \end{array}$$

2. Definitions

$$\begin{array}{rcl} p & ::= & C \\ C & ::= & \mathsf{skip} \mid x := e \mid C'; C'' \mid \mathsf{If}\ be\ \mathsf{Then}\ C'\ \mathsf{Else}\ C'' \mid \mathsf{While}\ be\ \mathsf{Do}\ C' \\ op & ::= & + \mid - \mid * \\ e & ::= & n \mid x \mid e'\ op\ e'' \\ bop & ::= & \mathit{And} \mid \mathit{Or} \\ be & ::= & bx \mid \mathsf{T} \mid \mathsf{F} \mid be'\ bop\ be'' \mid \mathsf{Not}\ be' \mid \mathsf{Equal}(e, e') \end{array}$$

Figure 12: Abstract syntax for *WhileL*.

```
< s(0) . mtS, (V('x),s(0)) . dec1, pop . mtC >
--->  *** rule Pop
< s(0) . mtS, dec1, mtC >

rewrites: 55 in 0ms cpu (152ms real) (~ rewrites/second)
result States: < s(0) . mtS, dec1, mtC >
```

4 The imperative language *WhileL*

In this section we present two operational semantics for a simple imperative programming language called *WhileL* in [36]. A program is a sequence of commands that can modify the memory, which is a collection of addresses where values are stored. As we have done for the functional language *Fpl*, we first describe the implementation of an evaluation semantics for *WhileL* and then the implementation of a computation semantics for the same language.

We also implement the semantics of the guarded command language *GuardL* in [36], which is a generalization of the language *WhileL* obtained by allowing non-determinism.

4.1 Imperative syntax definition

The abstract syntax for the language *WhileL* is shown in Figure 12, and it is implemented in the module WHILE-SYNTAX. Notice how the signature structure faithfully corresponds to the grammar structure defined by the abstract syntax of the language. As we did for *Fpl*, we add the definition of natural numbers.

```
fmod WHILE-SYNTAX is
  protecting QID .

  sorts Var Num Op Exp BVar Boolean BOp BExp Com Prog .

  op V : Qid -> Var .
  subsort Var < Exp .
  subsort Num < Exp .

  op 0 : -> Num .
  op s : Num -> Num .

  ops + - * : -> Op .

  op ___ : Exp Op Exp -> Exp [prec 20] .

  op BV : Qid -> BVar .
  subsort BVar < BExp .
```

$$\text{CR} \quad \frac{}{(\mathbf{n}, s) \Longrightarrow_A \mathbf{n}} \qquad\qquad \text{VarR} \quad \frac{}{(x, s) \Longrightarrow_A s(x)}$$

$$\text{OpR} \quad \frac{(e, s) \Longrightarrow_A v \qquad (e', s) \Longrightarrow_A v'}{(e\ op\ e', s) \Longrightarrow_A Ap(op, v, v')}$$

$$\text{BCR} \quad \frac{}{(\mathsf{T}, s) \Longrightarrow_B \mathsf{T}} \qquad\qquad \frac{}{(\mathsf{F}, s) \Longrightarrow_B \mathsf{F}}$$

$$\text{BVarR} \quad \frac{}{(bx, s) \Longrightarrow_B s(bx)}$$

$$\text{BOpR} \quad \frac{\begin{array}{c}(be, s) \Longrightarrow_B bv \\ (be', s) \Longrightarrow_B bv'\end{array}}{(be\ bop\ be', s) \Longrightarrow_B Ap(bop, bv, bv')}$$

$$\text{EqR} \quad \frac{\begin{array}{c}(e, s) \Longrightarrow_A v \\ (e', s) \Longrightarrow_A v\end{array}}{(\mathsf{Equal}(e, e'), s) \Longrightarrow_B \mathsf{T}} \qquad \frac{\begin{array}{c}(e, s) \Longrightarrow_A v \\ (e', s) \Longrightarrow_A v'\end{array}}{(\mathsf{Equal}(e, e'), s) \Longrightarrow_B \mathsf{F}}\ v \neq v'$$

$$\text{NotR} \quad \frac{(be, s) \Longrightarrow_B \mathsf{T}}{(\mathsf{Not}\ be, s) \Longrightarrow_B \mathsf{F}} \qquad\qquad \frac{(be, s) \Longrightarrow_B \mathsf{F}}{(\mathsf{Not}\ be, s) \Longrightarrow_B \mathsf{T}}$$

Figure 13: Evaluation semantics for *WhileL*, $\Longrightarrow_A$ and $\Longrightarrow_B$.

```
  subsort Boolean < BExp .

  ops T F : -> Boolean .
  ops And Or : -> BOp .
  op ___ : BExp BOp BExp -> BExp [prec 20] .
  op Not_ : BExp -> BExp [prec 15] .
  op Equal : Exp Exp -> BExp .

  op skip : -> Com .
  op _:=_ : Var Exp -> Com [prec 30] .
  op _;_ : Com Com -> Com [assoc prec 40] .
  op If_Then_Else_ : BExp Com Com -> Com [prec 50] .
  op While_Do_ : BExp Com -> Com [prec 60] .

  subsort Com < Prog .

endfm
```

4.2 Evaluation semantics

The evaluation semantics for *WhileL* is given by means of three relations: $\Longrightarrow_A$, $\Longrightarrow_B$, and $\Longrightarrow_C$, corresponding to each one of the syntactic categories *Exp* (arithmetic expressions), *BExp* (Boolean expressions), and *Com* (commands). Environments are also used to keep the value of variables. However, here the variables play a quite different role to that played in the functional language *Fpl*; now they represent memory addresses and, as said above, computation proceeds by modifying the contents of this memory. Therefore, although we reuse in the Maude code the same module `ENV`, we use the "memory" terminology in the text.

The evaluation relation $\Longrightarrow_A$ for arithmetic expressions takes a pair containing an expression and a memory, and it returns a value, the result of evaluating the expression in this memory. The same happens with the relation $\Longrightarrow_B$ for Boolean expressions. Their definitions are shown in Figure 13. The module `EVALUATION-EXP` implements both relations.

The sort `Statement` is used to describe the structure of a rule's lefthand side, consisting of a memory and an expression. Then `Num` and `Boolean` are made subsorts of `Statement`, thus ensuring that both sides of a rule have a common sort.

```
mod EVALUATION-EXP is
  protecting ENV .
  protecting AP .

  sort Statement .
  subsorts Num Boolean < Statement .

  op <_,_> : Exp ENV -> Statement .
  op <_,_> : BExp ENV -> Statement .

  var n : Num .
  var x : Var .
  var st : ENV .
  vars e e' : Exp .
  var op : Op .
  vars v v' : Num .
  var bx : BVar .
  vars bv bv' : Boolean .
  var bop : BOp .
  vars be be' : BExp .

  *** Evaluation semantics for expressions

  rl [CR] : < n, st > => n .

  rl [VarR] : < x, st > =>  st(x) .

  crl [OpR] : < e op e', st > => Ap(op,v,v')
           if < e, st > => v /\
              < e', st > => v' .

  rl [BCR1] : < T, st > => T .
  rl [BCR2] : < F, st > => F .

  rl [BVarR] : < bx, st > =>  st(bx) .

  crl [OpR] : < be bop be', st > => Ap(bop,bv,bv')
           if < be, st > => bv /\
              < be', st > => bv' .

  crl [EqR1] : < Equal(e,e'), st > => T
            if < e, st > => v /\
               < e', st > => v .
  crl [EqR2] : < Equal(e,e'), st > => F
            if < e, st > => v /\
               < e', st > => v' /\ v =/= v' .

  crl [Not1] : < Not be, st > => F
            if < be, st > => T .
  crl [Not2] : < Not be, st > => T
            if < be, st > => F .
endm
```

The evaluation relation for commands $\Longrightarrow_C$ takes a pair containing a command and a memory and it returns a new memory. Intuitively, the returned memory is the result of modifying the initial memory by means of the executed command. In this way, a judgement $(C, s) \Longrightarrow_C s'$ means that when the command C is executed on the memory s, the execution finishes and the final state of the memory is s'. The definition of the relation $\Longrightarrow_C$ is shown in Figure 14, and it is implemented in the module EVALUATION-WHILE.

```
mod EVALUATION-WHILE is
```

$$\text{SkipR} \quad \frac{}{(\mathsf{skip}, s) \Longrightarrow_C s} \qquad\qquad \text{AsR} \quad \frac{(e, s) \Longrightarrow_A v}{(x := e, s) \Longrightarrow_C s[v/x]}$$

$$\text{ComR} \quad \frac{\begin{array}{c}(C, s) \Longrightarrow_C s' \\ (C', s') \Longrightarrow_C s''\end{array}}{C; C' \Longrightarrow_C s''}$$

$$\text{IfR} \quad \frac{\begin{array}{c}(be, s) \Longrightarrow_B \mathsf{T} \\ (C, s) \Longrightarrow_C s'\end{array}}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C', s) \Longrightarrow_C s'} \qquad \frac{\begin{array}{c}(be, s) \Longrightarrow_B \mathsf{F} \\ (C', s) \Longrightarrow_C s'\end{array}}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C', s) \Longrightarrow_C s'}$$

$$\text{WhileR} \quad \frac{(be, s) \Longrightarrow_B \mathsf{F}}{(\mathsf{While}\ be\ \mathsf{Do}\ C, s) \Longrightarrow_C s} \qquad \frac{\begin{array}{c}(be, s) \Longrightarrow_B \mathsf{T} \\ (C'; \mathsf{While}\ be\ \mathsf{Do}\ C, s) \Longrightarrow_C s'\end{array}}{(\mathsf{While}\ be\ \mathsf{Do}\ C, s) \Longrightarrow_C s'}$$

Figure 14: Evaluation semantics for *WhileL*, $\Longrightarrow_C$.

```
  protecting EVALUATION-EXP .

  subsort ENV < Statement .

  op <_,_> : Com ENV -> Statement .

  var x : Var .
  vars st st' st'' : ENV .
  var e : Exp .
  var v : Num .
  var be : BExp .
  vars C C' : Com .


  *** Evaluation semantics for WhileL

  crl [AsR] : < x := e, st > => st[v / x]
           if < e, st > => v .

  rl [SkipR] : < skip, st > => st .

  crl [IfR1] : < If be Then C Else C', st > => st'
            if < be, st > => T /\
               < C, st > => st' .
  crl [IfR2] : < If be Then C Else C', st > => st'
            if < be, st > => F /\
               < C', st > => st' .

  crl [ComR] : < C ; C', st > => st''
            if < C, st > => st' /\
               < C', st' > => st'' .

  crl [WhileR1] : < While be Do C, st > => st
               if < be, st > => F .
  crl [WhileR2] : < While be Do C, st > => st'
               if < be, st > => T /\
                  < C ; (While be Do C), st > => st' .
endm
```

As an example of application of these rules let us consider the following program

$z := 0$;
$\mathsf{While\ Not\ (Equal}(x, 0)\mathsf{)\ Do}$
    $z := z + y$;

$$\text{AsRc} \quad \frac{(e,s) \Longrightarrow_A v}{(x := e, s) \longrightarrow_C (\mathsf{skip}, s[v/x])}$$

$$\text{ComRc} \quad \frac{(C,s) \longrightarrow_C (C'',s')}{(C;C',s) \longrightarrow_C (C'';C',s')} \qquad \frac{(C,s)\surd \qquad (C',s) \longrightarrow_C (C'',s')}{(C;C',s) \longrightarrow_C (C'',s')}$$

$$\text{IfRc} \quad \frac{(be,s) \Longrightarrow_B \mathsf{T} \qquad (C,s) \longrightarrow_C (C'',s')}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C',s) \longrightarrow_C (C'',s')} \qquad \frac{(be,s) \Longrightarrow_B \mathsf{F} \qquad (C',s) \longrightarrow_C (C'',s')}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C',s) \longrightarrow_C (C'',s')}$$

$$\text{WhileRc} \quad \frac{(be,s) \Longrightarrow_B \mathsf{F}}{(\mathsf{While}\ be\ \mathsf{Do}\ C,s) \longrightarrow_C (\mathsf{skip},s)}$$

$$\frac{(be,s) \Longrightarrow_B \mathsf{T}}{(\mathsf{While}\ be\ \mathsf{Do}\ C,s) \longrightarrow_C (C;\mathsf{While}\ be\ \mathsf{Do}\ C,s)}$$

Figure 15: Computation semantics for *WhileL*, $\longrightarrow_C$.

$$\text{Skipt} \quad \frac{}{(\mathsf{skip},s)\surd} \qquad \text{ComRt} \quad \frac{(C,s)\surd \qquad (C',s)\surd}{(C;C',s)\surd}$$

$$\text{IfRt} \quad \frac{(be,s) \Longrightarrow_B \mathsf{T} \qquad (C,s)\surd}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C',s)\surd} \qquad \frac{(be,s) \Longrightarrow_B \mathsf{F} \qquad (C',s)\surd}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C',s)\surd}$$

Figure 16: Termination predicate for *WhileL*.

$\qquad x := x - 1$

that calculates the product $x * y$ and saves the result in z. We execute it starting with a memory s where $s(x) = 2$, $s(y) = 3$, and $s(z) = 1$.

```
Maude> rew  <  V('z) := 0 ;
               (While Not Equal(V('x), 0) Do
                  V('z) := V('z) + V('y) ;
                  V('x) := V('x) - s(0)),
             V('x) = s(s(0)) V('y) = s(s(s(0))) V('z) = s(0) > .
rewrites: 267 in 0ms cpu (630ms real) (~ rewrites/second)
result ENV: V('x) = 0 V('y) = s(s(s(0))) V('z) = s(s(s(s(s(s(0))))))
```

4.3 Computation semantics

The basic commands in *WhileL* are assignments that modify the memory by changing the value associated to a variable. A computation semantics for *WhileL* has to describe the basic operations that each command can make, and in which order they are made. The judgement $(C,s) \longrightarrow_C (C',s')$ means that the command C can execute a basic operation that changes the memory from s to s', being C' the remainder of C that has still to be executed.

In this section we assume that we are not interested in how arithmetic and Boolean expressions are computed, so we do not define relations $\longrightarrow_A$ and $\longrightarrow_B$, using instead the evaluation relations $\Longrightarrow_A$ and $\Longrightarrow_B$ in the previous section. The rules defining the relation $\longrightarrow_C$ for commands are shown in Figure 15, where $(C,s)\surd$ indicates that the execution of command C has finished. The definition of this termination predicate is shown in Figure 16.

The implementation of these semantic rules is shown in the module `COMPUTATION-WHILE` below. The termination predicate has also been implemented by means of rules, that rewrite a pair containing a command and a memory to the constant `Tick`. Rewrite rules are needed instead of equations, because the predicate definition uses transitions in the premises of rules IfRt in Figure 16.

```
mod COMPUTATION-WHILE is
  protecting EVALUATION-EXP .

  op <_,_> : Com ENV -> Statement .

  sort Statement2 .
  op `(_,_`) : Com ENV -> Statement2 .
  op Tick : -> Statement2 .

  var x : Var .
  vars st st' : ENV .
  var e : Exp .
  var v : Num .
  var be : BExp .
  vars C C' C'' : Com .

  *** Computation semantics for WhileL

  crl [AsRc] : < x := e, st > => < skip, st[v / x] >
          if < e, st > => v .

  crl [IfRc1] : < If be Then C Else C', st > => < C'', st' >
           if < be, st > => T /\
              < C, st > => < C'', st' > /\ C =/= C'' .
  crl [IfRc2] : < If be Then C Else C', st > => < C'', st' >
           if < be, st > => F /\
              < C', st > => < C'', st' > /\ C' =/= C'' .

  crl [ComRc1] : < C ; C', st > => < C'' ; C', st' >
            if < C, st > => < C'', st' > /\ C =/= C'' .
  crl [ComRc2] : < C ; C', st > => < C'', st' >
            if ( C, st ) => Tick /\
               < C', st > => < C'', st' > /\ C' =/= C'' .

  crl [WhileRc1] : < While be Do C, st > => < skip, st >
              if < be, st > => F .
  crl [WhileRc2] : < While be Do C, st > => < C ; (While be Do C), st >
              if < be, st > => T .

  *** Termination predicate for WhileL

  rl [Skipt] : ( skip, st ) => Tick .

  crl [IfRt1] : ( If be Then C Else C', st ) => Tick
           if < be, st > => T /\
              ( C, st ) => Tick .
  crl [IfRt2] : ( If be Then C Else C', st ) => Tick
           if < be, st > => F /\
              ( C', st ) => Tick .

  crl [ComRt] : ( C ; C', st ) => Tick
           if ( C, st ) => Tick /\
              ( C', st ) => Tick .
endm
```

Note how in rule `IfRc1`, for example, the condition `C =/= C''` is used. This is needed here because the resolution of condition `< C, st > => < C'', st' >` means rewriting `< C, st >` zero or more times until the pattern `< C'', st' >` is matched. Since `C''` and `st'` are not bound before the condition is solved, the first attempt consists in the zero rewrites case, which matches `< C'', st' >`. To avoid this case (we want one step to be made) we require `C =/= C''`.

We can execute the same example program from the previous section.

```
Maude> rew  <  V('z) := 0 ;
```

1. Syntactic categories

$$\begin{array}{rcl@{\qquad\qquad}rcl} p & \in & \mathit{Prog} & e & \in & \mathit{Exp} \\ C & \in & \mathit{Com} & be & \in & \mathit{BExp} \\ GC & \in & \mathit{GuardCom} \end{array}$$

2. Definitions

$$\begin{array}{rcl} p & ::= & C \\ C & ::= & \mathsf{skip} \mid x := e \mid C';C'' \mid \mathsf{if}\ GC\ \mathsf{fi} \mid \mathsf{do}\ GC\ \mathsf{od} \\ GC & ::= & be \rightarrow C \mid GC' \ \Box\ GC'' \end{array}$$

Figure 17: Abstract syntax for *GuardL*.

```
              (While Not Equal(V('x), 0) Do
                 V('z) := V('z) + V('y) ;
                 V('x) := V('x) - s(0)),
            V('x) = s(s(0)) V('y) = s(s(s(0))) V('z) = s(0) > .
rewrites: 299 in 0ms cpu (826ms real) (~ rewrites/second)
result Statement:  < skip, V('x) = 0 V('y) = s(s(s(0))) V('z) = s(s(s(s(s(0)))))) >
```

The command `rew` applies rewrite rules (following its default strategy) until no more rule can be applied. The obtained result corresponds to the value returned by the evaluation semantics, since both semantics are deterministic.

Here we do not need to implement explicitly the reflexive, transitive closure of transition $\longrightarrow_C$, as we did in Section 3.3 for the computation semantics of *Fpl*, because now the lefthand and righthand sides of the rewrite rules have the same form, so the application of rules can be concatenated. But we can use the command `rew [`n`]` to see which is the obtained expression after the application of n rules. For example, after two applications of rules, variable `z` has been set to `0` and the loop has been unfolded once.

```
Maude> rew [2] < V('z) := 0 ;
                (While Not Equal(V('x),0) Do
                 V('z) := V('z) + V('y) ;
                 V('x) := V('x) - s(0)),
                V('x) = s(s(0)) V('y) = s(s(s(0))) V('z) = s(0) > .
rewrites: 70 in 0ms cpu (1ms real) (~ rewrites/second)
result Statement: < V('z) := V('z) + V('y) ;
                    V('x) := V('x) - s(0) ;
                    (While Not  Equal(V('x), 0) Do
                       V('z) := V('z) + V('y) ; V('x) := V('x) - s(0)),
                  V('x) = s(s(0)) V('y) = s(s(s(0))) V('z) = 0 >
```

4.4 The language *GuardL*

In [36] it is also presented a generalization of the language *WhileL* that allows non-determinism, called *GuardL*. This language was defined originally by Dijkstra in [24], where it was presented as a convenient language for the development of programs and their verification. Non-determinism was considered quite useful, since it allows the program designer to delegate some decisions that are next taken into account by the programmer or compiler.

The abstract syntax of the language *GuardL* is shown in Figure 17. A new syntactic category of *guarded commands* appears, whose elements have this general form:

$$be_1 \rightarrow C_1 \ \Box\ \ldots\ \Box\ be_k \rightarrow C_k.$$

The following module implements the syntax of *GuardL*, having the same structure of the corresponding grammar.

```
fmod GUARDL-SYNTAX is
  protecting QID .

  sorts Var Num Op Exp BVar Boolean BOp BExp Com GuardCom Prog .

  op V : Qid -> Var .
  subsort Var < Exp .
  subsort Num < Exp .

  op z : -> Num .
  op s : Num -> Num .

  ops + - * : -> Op .

  op ___ : Exp Op Exp -> Exp [prec 20] .

  op BV : Qid -> BVar .
  subsort BVar < BExp .
  subsort Boolean < BExp .

  ops T F : -> Boolean .
  ops And Or : -> BOp .
  op ___ : BExp BOp BExp -> BExp [prec 20] .
  op Not_ : BExp -> BExp [prec 15] .
  op Equal : Exp Exp -> BExp .

  op skip : -> Com .
  op _:=_ : Var Exp -> Com [prec 30] .
  op _;_ : Com Com -> Com [assoc prec 40] .
  op if_fi : GuardCom -> Com [prec 50] .
  op do_od : GuardCom -> Com [prec 60] .

  op _->_ : BExp Com -> GuardCom [prec 42] .
  op _[]_ : GuardCom GuardCom -> GuardCom [assoc prec 45] .

  subsort Com < Prog .
endfm
```

In a guarded command, the Boolean expression be_i *guards* the corresponding command C_i, which will be executed only if the control "goes through the guard be_i", that is, be_i is evaluated to true. In the command if GC fi, only a command associated to a true guard will be executed. If no guard is true, it is considered that an execution error has been produced. In the same way, the command do GC od is a generalization of the command While. The guarded command GC is executed in a repeated way, while at least one of the guards is true. Termination takes place when all the guards are false. These intuitive ideas are formalized in the computation semantics in Figure 18, that defines two transition relations, $\longrightarrow_C$ and $\longrightarrow_{GC}$, and uses the evaluation semantics in Figure 13 for arithmetic and Boolean expressions.

To formalize the fact that all the Boolean guards of a guarded command are false, a failure predicate is needed. This is defined in an inductive way in Figure 19. A termination predicate is also used, like in the case of *WhileL*, although now it is even simpler, and it is defined in Figure 20.

The following module GUARDL-COMPUTATION implements the computation semantics for the language *GuardL*.

```
mod GUARDL-COMPUTATION is
  protecting ENV .
  protecting AP .
  protecting EVALUATION-EXP .

  op <_,_> : Com ENV -> Statement .
  op <_,_> : GuardCom ENV -> Statement .
```

AsRc $$\frac{(e, s) \Longrightarrow_A v}{(x := e, s) \longrightarrow_C (\mathsf{skip}, s[v/x])}$$

ComRc $$\frac{(C, s) \longrightarrow_C (C'', s')}{(C; C', s) \longrightarrow_C (C''; C', s')} \qquad \frac{(C, s)\surd \qquad (C', s) \longrightarrow_C (C'', s')}{(C; C', s) \longrightarrow_C (C'', s')}$$

IfRc $$\frac{(GC, s) \longrightarrow_{GC} (C, s)}{(\mathsf{if}\ GC\ \mathsf{fi}, s) \longrightarrow_C (C, s)}$$

DoRc $$\frac{(GC, s) \longrightarrow_{GC} (C, s)}{(\mathsf{do}\ GC\ \mathsf{od}, s) \longrightarrow_C (C; \mathsf{do}\ GC\ \mathsf{od}, s)} \qquad \frac{(GC, s)\ \mathit{fails}}{(\mathsf{do}\ GC\ \mathsf{od}, s) \longrightarrow_C (\mathsf{skip}, s)}$$

GCRc $$\frac{(be, s) \Longrightarrow_B \mathsf{T}}{(be \rightarrow C, s) \longrightarrow_{GC} (C, s)}$$

$$\frac{(GC_1, s) \longrightarrow_{GC} (C, s)}{(GC_1 \,\square\, GC_2, s) \longrightarrow_{GC} (C, s)} \qquad \frac{(GC_2, s) \longrightarrow_{GC} (C, s)}{(GC_1 \,\square\, GC_2, s) \longrightarrow_{GC} (C, s)}$$

Figure 18: Computation semantics for *GuardL*, $\longrightarrow_C$ and $\longrightarrow_{GC}$.

IfRf1 $$\frac{(be, s) \Longrightarrow_B \mathsf{F}}{(be \rightarrow C, s)\ \mathit{fails}}$$ IfRf2 $$\frac{(GC, s)\ \mathit{fails} \qquad (GC', s)\ \mathit{fails}}{(GC \,\square\, GC', s)\ \mathit{fails}}$$

Figure 19: Failure predicate for *GuardL*.

Skipt $$\frac{}{(\mathsf{skip}, s)\surd}$$ ComRt $$\frac{(C, s)\surd \qquad (C', s)\surd}{(C; C', s)\surd}$$

Figure 20: Termination predicate for *GuardL*.

```
  sort Statement2 .
  op `(_,_`) : Com ENV -> Statement2 .
  op Tick : -> Statement2 .
  op `(_,_`) : GuardCom ENV -> Statement2 .
  op fails : -> Statement2 .

  var x : Var .
  vars st st' : ENV .
  var e : Exp .
  var op : Op .
  vars v v' : Num .
  var be : BExp .
  vars C C' C'' : Com .
  vars GC GC' : GuardCom .

  *** Computation semantics for GuardL

  crl [AsRc] : < x := e, st > => < skip, st[v / x] >
          if < e, st > => v .

  crl [ifRc] : < if GC fi, st > => < C, st >
          if < GC, st > => < C, st > .

  crl [ComRc1] : < C ; C', st > => < C'' ; C', st' >
            if < C, st > => < C'', st' > /\ C =/= C'' .
  crl [ComRc2] : < C ; C', st > => < C'', st' >
            if ( C, st ) => Tick /\
               < C', st > => < C'', st' > /\ C' =/= C'' .

  crl [doRc1] : < do GC od, st > => < C ; (do GC od), st >
           if < GC, st > => < C, st > .
  crl [doRc2] : < do GC od, st > => < skip, st >
           if ( GC, st ) => fails .


  crl [GCRc1] : < be -> C, st > => < C, st >
           if < be, st > => T .

  crl [GCRc2] : < GC [] GC', st > => < C, st >
           if < GC, st > => < C, st > .
  crl [GCRc2] : < GC [] GC', st > => < C, st >
           if < GC', st > => < C, st > .

  *** Failure predicate for GuardL

  crl [IfRf1] : ( be -> C, st ) => fails
           if < be, st > => F .

  crl [IfRf2] : ( GC [] GC', st ) => fails
           if ( GC, st )  => fails /\
              ( GC', st ) => fails .

  *** Termination predicate for GuardL

  rl [Skipt] : ( skip, st ) => Tick .

  crl [ComRt] : ( C ; C', st ) => Tick
           if ( C, st ) => Tick /\
              ( C', st ) => Tick .
endm
```

Non-determinism appears by means of rules `GCRc2` and `GCRc3`. They both have the same lefthand side, so when one of them can be applied, also the other one can, assuming both `GC` and "GC'" have a true guard.

The failure and termination predicates have been also implemented by means of rules, as in the previous section. However, in this case, it is not necessary to use rewrite rules to define the termination predicate, since the only rewrites in the conditions refer to the own predicate. In this way, we can also define this predicate by means of a Boolean operation. Notice the use of the `owise` attribute to define the predicate in all the cases in an easy way.

```
op (_,_)Tick : Com ENV -> Bool .
eq ( skip, st )Tick = true .
ceq ( C ; C', st )Tick = true
 if ( C, st )Tick  /\  ( C', st )Tick .
eq ( C, st )Tick = false [owise] .
```

When using this predicate, the rule `ComRc2` has to be modified in the following way:

```
crl [ComRc2] : < C ; C', st > => < C'', st' >
            if ( C, st )Tick /\ < C', st > => < C'', st' > /\ C' =/= C'' .
```

To illustrate this semantics we can execute the following command, borrowed from [36, page 133]:

$$\begin{array}{l} \mathsf{do} \\ \quad x > 0 \rightarrow x := x - 1 \;;\; y := y + 1 \\ \quad \Box \\ \quad x > 2 \rightarrow x := x - 2 \;;\; y := y + 1 \\ \mathsf{od} \end{array}$$

If we start to execute this command in a memory s in which $s(x) = 5$ and $s(y) = 0$, and assuming appropriate rules for the operation $>$, three different states can be reached. To ask Maude to show all the final reachable states, we use the command `search` (with version `=>+` to allow several steps to find the desired term).

```
Maude> search
      < do  V('x) > 0       -> V('x) := V('x) - s(0) ; V('y) := V('y) + s(0)
         [] V('x) > s(s(0)) -> V('x) := V('x) - s(s(0)) ; V('y) := V('y) + s(0)
        od,
      V('x) = s(s(s(s(s(0))))) V('y) = 0 >  =>+  < skip, st:ENV > .
Solution 1 (state 25)
   st --> V('x) = 0 V('y) = s(s(s(s(s(0)))))
Solution 2 (state 29)
   st --> V('x) = 0 V('y) = s(s(s(s(0))))
Solution 3 (state 38)
   st --> V('x) = 0 V('y) = s(s(s(0)))
No more solutions.
```

5 Mini-ML

In this section we implement the evaluation semantics (or natural semantics) for the functional language Mini-ML as described by Kahn in [45]. The abstract syntax, such as it is defined in [45], is presented in Figure 21. Notice how this syntax presentation is closer to the signature of an algebraic specification. The syntax defines a λ-calculus extended with products, if, let, and letrec. In an expression λP.E, P is a pattern, which is either an identifier, like the variable x, or a pair of patterns $(\mathrm{P}_1, \mathrm{P}_2)$, like the pattern $(x, (y, z))$.

The Mini-ML syntax is implemented by means of the following module:

sorts

EXP, IDENT, PAT, NULLPAT

subsorts

EXP $\supset$ NULLPAT, IDENT   PAT $\supset$ NULLPAT, IDENT

constructors

Patterns

pairpat	:	PAT×PAT	$\rightarrow$	PAT
nullpat	:		$\rightarrow$	NULLPAT

Expressions

number	:		$\rightarrow$	EXP
false	:		$\rightarrow$	EXP
true	:		$\rightarrow$	EXP
ident	:		$\rightarrow$	IDENT
lambda	:	PAT×EXP	$\rightarrow$	EXP
if	:	EXP×EXP×EXP	$\rightarrow$	EXP
mlpair	:	EXP×EXP	$\rightarrow$	EXP
apply	:	EXP×EXP	$\rightarrow$	EXP
let	:	PAT×EXP×EXP	$\rightarrow$	EXP
letrec	:	PAT×EXP×EXP	$\rightarrow$	EXP

Figure 21: Abstract syntax for Mini-ML.

```
fmod MINI-ML-SYNTAX is
  protecting QID .

  sort Nats TruthVal Var .

  op 0 : -> Nats .
  op s : Nats -> Nats .

  ops true false : -> TruthVal .

  op id : Qid -> Var .

  sorts Exp Value Pat NullPat Lambda .

  subsorts NullPat Var < Pat .
  op () : -> NullPat .
  op (_,_) : Pat Pat -> Pat .

  op (_,_) : Var Var -> Var .

  subsorts TruthVal Nats < Value .
  op (_,_) : Value Value -> Value .

  subsorts Value Var Lambda < Exp .
  op s : Exp -> Exp .
  op _+_ : Exp Exp -> Exp [prec 20] .
  op not : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
  op if_then_else_ : Exp Exp Exp -> Exp [prec 22] .
  op (_,_) : Exp Exp -> Exp .
  op __ : Exp Exp -> Exp [prec 20] .
  op \_._ : Pat Exp -> Lambda [prec 15] .
  op let_=_in_ : Pat Exp Exp -> Exp [prec 25] .
  op letrec_=_in_ : Pat Exp Exp -> Exp [prec 25] .
endfm
```

The Mini-ML semantics is defined in [45] by means of judgements of the form $\rho \vdash \textsc{e} \Rightarrow \alpha$, where E in a Mini-ML expression, ρ is an environment, and α is the result of the evaluation of E in ρ. Functions

$$\rho \vdash \text{number N} \Rightarrow \text{N} \qquad \rho \vdash \text{true} \Rightarrow \mathit{true} \qquad \rho \vdash \text{false} \Rightarrow \mathit{false}$$

$$\rho \vdash \lambda \mathrm{P.E} \Rightarrow [\![\lambda \mathrm{P.E}, \rho]\!] \qquad \frac{\rho \overset{\text{val\_of}}{\vdash} \text{ident I} \mapsto \alpha}{\rho \vdash \text{ident I} \Rightarrow \alpha}$$

$$\frac{\rho \vdash \mathrm{E}_1 \mapsto \mathit{true} \qquad \rho \vdash \mathrm{E}_2 \Rightarrow \alpha}{\rho \vdash \text{if } \mathrm{E}_1 \text{ then } \mathrm{E}_2 \text{ else } \mathrm{E}_3 \Rightarrow \alpha} \qquad \frac{\rho \vdash \mathrm{E}_1 \mapsto \mathit{false} \qquad \rho \vdash \mathrm{E}_3 \Rightarrow \alpha}{\rho \vdash \text{if } \mathrm{E}_1 \text{ then } \mathrm{E}_2 \text{ else } \mathrm{E}_3 \Rightarrow \alpha}$$

$$\frac{\rho \vdash \mathrm{E}_1 \Rightarrow \alpha \qquad \rho \vdash \mathrm{E}_2 \Rightarrow \beta}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) \Rightarrow (\alpha, \beta)}$$

$$\frac{\rho \vdash \mathrm{E}_1 \Rightarrow [\![\lambda \mathrm{P.E}, \rho_1]\!] \qquad \rho \vdash \mathrm{E}_2 \Rightarrow \alpha \qquad \rho \cdot \mathrm{P} \mapsto \alpha \vdash \mathrm{E} \Rightarrow \beta}{\rho \vdash \mathrm{E}_1 \ \mathrm{E}_2 \Rightarrow \beta}$$

$$\frac{\rho \vdash \mathrm{E}_2 \Rightarrow \alpha \qquad \rho \cdot \mathrm{P} \mapsto \alpha \vdash \mathrm{E}_1 \Rightarrow \beta}{\rho \vdash \text{let } \mathrm{P} = \mathrm{E}_2 \text{ in } \mathrm{E}_1 \Rightarrow \beta}$$

$$\frac{\rho \cdot \mathrm{P} \mapsto \alpha \vdash \mathrm{E}_2 \Rightarrow \alpha \qquad \rho \cdot \mathrm{P} \mapsto \alpha \vdash \mathrm{E}_1 \Rightarrow \beta}{\rho \vdash \text{letrec } \mathrm{P} = \mathrm{E}_2 \text{ in } \mathrm{E}_1 \Rightarrow \beta}$$

Figure 22: Evaluation semantics for Mini-ML.

are handled like any other value; for example, they can be passed as parameters to other functions, or returned as the value on an expression.

Semantic values are:

- integer values;
- Boolean values *true* and *false*;
- closures like $[\![\lambda \mathrm{P.E}, \rho]\!]$, where P is a pattern, E is an expression, and ρ is an environment; and
- pairs of semantic values of the form (α, β), where of course α and β can also be pairs, giving rise to trees of semantics values.

The semantic rules of Mini-ML, such as they are presented in [45], are shown in Figure 22.

The implementation of the environments and the semantic rules, except for the operator letrec which is problematic and will be treated afterwards, is quite straightforward following the ideas already described in the previous sections and is shown in the following modules. Again, the sort `Statement` is used to describe the structure in each side of a rule, including as before the environment in the lefthand side. The operation `_|-val-of_` is used to obtain the value associated to an identifier in an environment.

```
fmod ENV is
  including MINI-ML-SYNTAX .
  sort Pair .
  op _->_ : Pat Value -> Pair [prec 10] .
  sort Env .
  subsort Pair < Env .
  op nil : -> Env .
  op _*_ : Env Env -> Env [assoc id: nil prec 20] .
  op Clos : Lambda Env -> Value .
endfm

mod MINI-ML-SEMANTICS is
  including ENV .

  sort Statement .
  op _|-_ : Env Exp -> Statement [prec 40] .
  op _|-val-of_ : Env Var -> Statement [prec 40] .

  subsort Value < Statement .

  vars RO RO1 : Env .
  vars N M : Nats .
  vars P P1 P2 : Pat .
  vars E E1 E2 E3 : Exp .
  vars I X : Qid .
  vars A B C : Value .

  rl [number] : RO |- N => N .

  crl [add] : RO |- E1 + E2 => C  if
              RO |- E1 => A  /\  RO |- E2 => B  /\  C := sum(A,B) .

  op sum : Nats Nats -> Nats .
  eq sum(0,N) = N .
  eq sum(s(N),M) = s(sum(N,M)) .

  rl [true] : RO |- true => true .
  rl [false] : RO |- false => false .

  rl [lambda] : RO |- \ P . E => Clos( \ P . E, RO) .

  crl [id] : RO |- id(I) => A
          if RO |-val-of id(I) => A .

  crl [if] : RO |- if E1 then E2 else E3 => A if
             RO |- E1 => true /\ RO |- E2 => A .
  crl [if] : RO |- if E1 then E2 else E3 => A if
             RO |- E1 => false /\ RO |- E3 => A .

  crl [pair] : RO |- ( E1, E2 ) => ( A, B ) if
               RO |- E1 => A  /\  RO |- E2 => B .

  crl [app] : RO |- E1 E2 => B if
              RO |- E1 => Clos( \ P . E, RO1)  /\
              RO |- E2 => A  /\
             (RO1 * P -> A ) |- E => B .

  crl [let] : RO |- let P = E2 in E1 => B if
              RO |- E2 => A  /\  (RO * P -> A) |- E1 => B .

  *** set VAL_OF

  rl [val_of] : RO * id(I) -> A |-val-of id(I) => A .

  crl [val_of] : RO * id(X) -> B |-val-of id(I) => A
              if X =/= I /\ RO |- id(I) => A .

  crl [val_of] : RO * (P1, P2) -> (A, B) |-val-of id(I) => C
              if RO * P1 -> A * P2 -> B  |-val-of id(I) => C .
endm
```

We evaluate now some of the expressions used as examples in [45]. For example, we can illustrate the block structure of the language and the use of patterns with the following expression, which is evaluated to 3.

$$\begin{array}{l}\mathsf{let}\ (x,y) = (2,3)\\ \mathsf{in}\ \mathsf{let}\ (x,y) = (y,x)\ \mathsf{in}\ x\end{array}$$

```
Maude> rew nil |- let ( id('x), id('y) ) = ( s(s(0)), s(s(s(0))) )
                  in let ( id('x), id('y) ) = ( id('y), id('x) ) in id('x) .
result Nats: s(s(s(0)))
```

We can also use higher-order functions, like in

$$\begin{array}{l}\mathsf{let}\ succ = \lambda x.x+1\\ \mathsf{in\ let}\ twice = \lambda f.\lambda x.(f(f\ x))\\ \quad \mathsf{in}\ ((twice\ succ)\ 0)\end{array}$$

```
Maude> rew
 nil |- let id('succ) = (\ id('x) . (id('x) + s(0)))
       in let id('twice) = (\ id('f) . (\ id('x) . (id('f) (id('f) id('x)))))
          in (( id('twice) id('succ)) 0) .
result Nats: s(s(0))
```

However, the semantic rule for the operator letrec cannot be implemented in a direct way, since the rule

```
  crl [letrec] : RO |- letrec P = E2 in E1 => B if
               (RO * P -> A) |- E2 => A  /\  (RO * P -> A) |- E1 => B .
```

is not admissible due to the fact that the variable A in the first rewrite condition is a new variable that appears both in the lethand side and in the righthand side of the rewrite condition, and thus its value cannot be obtained by matching.

This requires to modify the (textbook) presentation of the semantic rule that defines this operator. First, if we work with call-by-value, as we are doing here, it only makes sense to allow recursive definitions of λ-abstractions, because this kind of values is the only one for which we can guarantee termination since a λ-abstraction evaluates directly to a closure. For this case, Reynolds presents in [58, page 230] the following rule for letrec with only one variable as parameter:

$$\frac{\rho \vdash (\lambda v\,.\,e)(\lambda u\,.\,\text{letrec } v = \lambda u\,.\,e' \text{ in } e') \Rightarrow \alpha}{\rho \vdash \text{letrec } v = \lambda u\,.\,e' \text{ in } e \Rightarrow \alpha}$$

The intuitive idea is that in the premise the recursive definition has been *unfolded once*. Under call-by-value, the argument $(\lambda u\,.\,\text{letrec } v = \lambda u\,.\,e' \text{ in } e')$ can only be evaluated if it is a function, which is then evaluated to a closure.

We have generalized this rule to the Mini-ML case, where we can have definitions with patterns:

$$\frac{\rho \vdash (\lambda \mathrm{P}\,.\,\mathrm{E}')\ \mathrm{E}^* \Rightarrow \alpha}{\rho \vdash \text{letrec } \mathrm{P} = \mathrm{E} \text{ in } \mathrm{E}' \Rightarrow \alpha}$$

where P and E have the same form (regarding nesting), E contains only λ-abstractions, and E^* has the same form as E except that the body of each function has been substituted by a letrec, as we have done in the previous (simple) case. The implementation in Maude, which is equivalent to the corresponding formal definition of E^*, is the following one:

```
  op e* : Pat Exp Exp -> Exp .

  eq e*(P, E, \ P' . E1) = \ P' . (letrec P = E in E1) .
  eq e*(P, E, ( E1, E2 )) = ( e*(P, E, E1), e*(P, E, E2) ) .

  crl [letrec] : RO |- letrec P = E in E' => A
             if RO |- (\ P . E') e*(P, E, E) => A .
```

Now we are ready to evaluate other examples in [45]:

$$\begin{array}{lcl}\mathsf{letrec}\ (even, odd) & = & (\lambda x\,.\,\mathsf{if}\ x=0\ \mathsf{then}\ \text{true}\ \mathsf{else}\ odd(x-1),\\ & & \ \lambda x\,.\,\mathsf{if}\ x=0\ \mathsf{then}\ \text{false}\ \mathsf{else}\ even(x-1))\\ \mathsf{in}\ even(3). & & \end{array}$$

```
Maude> rew
    nil |- letrec (id('even), id('odd)) =
               ( (\ id('x) . (if (id('x) = 0) then true
                              else (id('odd) (id('x) - s(0))))),
                 (\ id('x) . (if (id('x) = 0) then false
                              else (id('even) (id('x) - s(0)))))
               )
           in id('even) s(s(s(0))) .
result TruthVal: false
```

6 CCS

In this section we describe in detail an implementation of the structural operational semantics for Milner's *Calculus of Communicating Systems*, CCS [52]. The main novelty with respect to the implementation of the previous languages is the use of the `frozen` attribute. We will also see how Maude can be used to implement other kinds of semantics, as illustrated with the semantics of the Hennessy-Milner logic [38] on top of CCS; however, in this logic example we will need to use also Maude's metalevel.

First, we provide a very brief introduction to CCS. We assume a set A of *names*; the elements of the set $\overline{A} = \{\overline{a} \mid a \in A\}$ are called *co-names*, and the members of the (disjoint) union $\mathcal{L} = A \cup \overline{A}$ are *labels* naming ordinary actions. The function $a \mapsto \overline{a}$ is extended to $\mathcal{L}$ by defining $\overline{\overline{a}} = a$. There is a special action called *silent action* and denoted τ, intended to represent internal behaviour of a system, and in particular the synchronization of two processes by means of complementary actions a and $\overline{a}$. Then the set of *actions* is $\mathcal{L} \cup \{\tau\}$. The set of processes is intuitively defined as follows:

- 0 is the inactive process that does nothing.
- If α is an action and P is a process, $\alpha.P$ is the process that performs α and subsequently behaves as P.
- If P and Q are processes, $P + Q$ is the process that may behave as either P or Q.
- If P and Q are processes, $P \mid Q$ represents P and Q running concurrently with possible communication via synchronization of a pair of ordinary actions a and $\overline{a}$.
- If P is a process and $f : \mathcal{L} \to \mathcal{L}$ is a (finite) relabelling function such that $f(\overline{a}) = \overline{f(a)}$, $P[f]$ is the process that behaves as P but with the actions relabelled according to f, assuming $f(\tau) = \tau$.
- If P is a process and $L \subseteq \mathcal{L}$ is a (finite) set of ordinary actions, $P \backslash L$ is the process that behaves as P but with the actions in $L \cup \overline{L}$ prohibited.
- If P is a process, X is a process identifier, and $X =_{def} P$ is a defining equation where P may recursively involve X, then X is a process that behaves as P.

This intuitive explanation can be made precise in terms of the structural operational semantics shown in Figure 23, that defines a labelled transition system for CCS processes. To simplify the presentation, we have already assumed that the operators for summation and parallel composition are commutative and associative, thus using a more abstract syntax and eliminating the need for symmetric cases in the corresponding rules.

6.1 CCS syntax

We define the CCS syntax in Maude. Quoted identifiers are used to represent labels and process identifiers. Notice the attributes `assoc` and `comm` for the summation and parallel composition operators. All the non-constant operators for building processes have been defined as *frozen*; we explain the reason for this in Section 6.2.

$$\text{Pref}\quad \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{Sum}\quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{Res}\quad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$\text{Par}\quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\text{Rel}\quad \frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L}\ \alpha \notin L \cup \overline{L} \qquad \text{Def}\quad \frac{P \xrightarrow{\alpha} P'}{X \xrightarrow{\alpha} P'}\ X =_{def} P$$

Figure 23: CCS operational semantics rules.

```
fmod CCS-SYNTAX is
  including QID .
  sorts Label Act ProcessId Process .
  subsorts Qid < Label < Act .
  subsorts Qid < ProcessId < Process .
  op ~_ : Label -> Label .
  eq ~ ~ L:Label = L:Label .
  op tau : -> Act .
  op 0 : -> Process .
  op _._ : Act Process -> Process [frozen prec 25] .
  op _+_ : Process Process -> Process [frozen assoc comm prec 35] .
  op _|_ : Process Process -> Process [frozen assoc comm prec 30] .
  op _[_/_] : Process Label Label -> Process [frozen prec 20] .
  op _\_ : Process Label -> Process [frozen prec 20] .
endfm
```

We represent full CCS, including (possibly recursive) process definitions by means of *contexts*. We have defined these contexts together with operations to work with them in the module `CCS-CONTEXT` below. It includes a constant `context` used to keep the definitions of the process identifiers used in each CCS specification.

```
fmod CCS-CONTEXT is  including CCS-SYNTAX .
  sort Context .
  op _=def_ : ProcessId Process -> Context [prec 40] .
  op nil : -> Context .
  op _&_ : [Context] [Context] -> [Context] [assoc comm id: nil prec 42] .
  op _definedIn_ : ProcessId Context -> Bool .
  op def : ProcessId Context -> [Process] .
  op context : -> Context .
  vars X X' : ProcessId .
  var P : Process .
  vars C C' : Context .
  cmb (X =def P) & C : Context if not(X definedIn C) .
  eq X definedIn nil = false .
  eq X definedIn (X' =def P & C') = (X == X') or (X definedIn C') .
  eq def(X, (X' =def P) & C') = if X == X' then P else def(X, C') fi .
endfm
```

Notice how the union of contexts `_&_` is a partial operation defined at the level of kinds (`[Context]`). The union of two contexts is a correct context if the defined process identifiers are disjoint. The (conditional) membership axiom (`cmb`) establishes this fact, using the auxiliary operation `_definedIn_`.

6.2 Implementation of CCS operational semantics

In order to implement the CCS semantics in Maude with transitions as rewrites, we want to interpret a CCS transition $P \xrightarrow{a} P'$ as a rewriting logic rewrite. However, rewrites have no labels, which are essential in the CCS semantics; therefore, we instead make the label a part of the resulting term, obtaining in this way a rewrite of the form $P \longrightarrow \{a\}P'$, where $\{a\}P'$ is a value of sort `ActProcess`, a supersort of `Process`. The following module, which is an admissible module [15] and therefore directly executable, includes the CCS semantics implementation.

```
mod CCS-SEMANTICS is
  protecting CCS-CONTEXT .
  sort ActProcess .
  subsort Process < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess [frozen] .
  vars L M : Label .            var A : Act .
  vars P P' Q Q' : Process .    var X : ProcessId .
  *** Prefix
  rl [Pref] : A . P => {A}P .
  *** Summation
  crl [Sum] : P + Q => {A}P' if P => {A}P' .
  *** Composition
  crl [Par1] : P | Q => {A}(P' | Q) if P => {A}P' .
  crl [Par2] : P | Q => {tau}(P' | Q') if P => {L}P' /\ Q => {~ L}Q' .
  *** Relabelling
  crl [Rel1] : P[M / L] => {M}(P'[M / L]) if P =>{L}P' .
  crl [Rel2] : P[M / L] => {~ M}(P'[M / L]) if P =>{~ L}P' .
  crl [Rel3] : P[M / L] => {A}(P'[M / L]) if P =>{A}P' /\ A =/= L /\ A =/= ~ L .
  *** Restriction
  crl [Res] : P \ L => {A}(P' \ L) if P => {A}P' /\ A =/= L /\ A =/= ~ L .
  *** Definition
  crl [Def] : X => {A}P if (X definedIn context) /\ def(X,context) => {A}P .
endm
```

In this semantic representation, the rewrite rules have the property of being sort-increasing, i.e., in a rewrite $t \longrightarrow t'$, the least sort of t' is bigger than the least sort of t. If we restrict ourselves to terms that are well formed in the sense that they can be assigned a sort (and not only a kind), one rule cannot be applied unless the resulting term is well formed, that is, it has a sort. For example, although `A . P` $\longrightarrow$ `{A}P` is a correct transition, we cannot derive `(A . P) | Q` $\longrightarrow$ `({A}P) | Q` because the righthand side term is not well formed. In this way, rewrites are only allowed to happen at the top of a process term, and not inside the term.

However, this mechanism to block undesired sort-increasing rewrites is not implemented in the current Maude 2.0 system, because term rewriting can happen at the level of kinds, and not only at the level of sorts. Therefore, our solution has been to declare all the syntax operators as `frozen`, which prevents the arguments of the corresponding operators from being rewritten by rules; see module `CCS-SYNTAX` in Section 6.1. This has not been necessary in the previous sections, because there all the additional structure (like environments) were put in the lefthand side of the rules and this directly disallowed the application of rewrite rules inside terms (that is, the congruence rule of rewriting logic could not be applied), as well as the concatenation of rewrites (that is, the transitivity rule could not be used).

Moreover, in the presence of rewrite conditions and infinite processes (those with an infinite number of successors), the `frozen` attribute will become much more useful, as we are going to see right now. If we have the rewrite condition `P => {A}Q`, and assume that the attribute `frozen` is not used, then `P` is tried to be rewritten in any possible way, and the result is matched against the pattern `{A}Q`. For instance, if in a correct application of this rule `P` is of the form `(A . P') | Q'`, then `P` is rewritten to `({A}P') | Q'` although then the result is rejected. The problem appears when we have recursive processes, because the built-in search that tries to satisfy the rewrite condition can become infinite and not terminate. For example, if `P'` above is recursive, given by `P' = A . P'`, then `P` is rewritten to `({A}P') | Q'`, `({A}{A}P') | Q'`, etc.,

$$\begin{tikzcd}
 & P \arrow[dl] \arrow[d] \arrow[dr] & & \cdots \\
(\{A\}P') \mid Q' & (\{A\}\{A\}P') \mid Q' & (\{A\}\{A\}\{A\}P') \mid Q' &
\end{tikzcd}$$

although all these results are going to be rejected because they are not well formed. Moreover, these rewrites do not correspond to any transition in the CCS semantics, where transitions always occur at the top of a process.

Although the `frozen` attribute solves the previous problem, it still appears when we want to know *all* the possible rewrites of the above process `P'` which are of the form `{A}Q` with `Q` of sort `Process` (as we do in Section 6.4 to implement the modal logic semantics). In this case, `P'` is rewritten to `{A}P'`, but also to `{A}{A}P'`, `{A}{A}{A}P'`, etc.,

$$\begin{tikzcd}
 & & \texttt{P'} \arrow[dll] \arrow[dl] \arrow[d] & \cdots \\
\texttt{\{A\}P'} & \texttt{\{A\}\{A\}P'} & \texttt{\{A\}\{A\}\{A\}P'} &
\end{tikzcd}$$

and only the first rewrite matches the pattern `{A}Q`. Thus, we have to declare also the operator `{_}_` as frozen.

In summary, we use the `frozen` attribute to ensure that rewrites happen only at the top as well as to avoid an infinite loop in the search process when we know that the search would be unsuccessful, although the search may be unsuccessful for two different reasons: either because the built terms are not well formed, as in `({A}P') | Q'`, and that is the reason why the syntax operators are frozen; or because the terms do not match the given pattern, as in `{A}{A}P'`, and that is the reason why `{_}_` is frozen.

A disadvantage is that with the shape of rewrite rules in `CCS-SEMANTICS` and all the constructor operators being declared as frozen, we have lost the ability of proving that a process can perform a sequence of actions, or *trace*, because the rules can only be used to obtain *one-step* successors. The congruence rule of rewriting logic cannot be used because the operators are frozen, and the transitivity rule cannot be used because all the rules rewrite to something of the form `{A}Q`, and there is no rule with this pattern in the lefthand side. This is not a problem if we want to use the semantics only in the definition of the modal logic semantics, because there only one-step successors are needed.

However, we can solve this by extending the semantics with rules that generate the transitive closure of the CCS transitions as follows:

```
sort TProcess .
subsort TProcess < ActProcess .
op [_] : Process -> TProcess [frozen] .
crl [refl] : [ P ] => {A}Q  if P => {A}Q .
crl [tran] : [ P ] => {A}AP if P => {A}Q /\ [ Q ] => AP .
```

Notice how we use the dummy operator `[_]`. If we did not use it in the lefthand side of the above rules, the lefthand side of both the head of the rule and the rewrites in conditions would be variables that match any term and then the rule itself could be used in order to solve its first condition, giving rise to an infinite loop. In addition, the dummy operator has also been declared as frozen in order to avoid useless rewrites like for example `[ A . P ]` $\longrightarrow$ `[ {A}P ]`. This dummy operator is used to control which rules we want to be applied to resolve the conditions in rules `refl` and `tran`. This is a similar technique to that used in Section 3.3 when a different operator `_,_|=_` was used to represent the reflexive, transitive closure of the transition relation $\longrightarrow_A$.

The resulting representation of CCS, with these two last rules, is semantically correct in the sense that given a CCS process P, there are processes $P_1, \ldots, P_k$ such that

$$P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} P_k$$

if and only if `[ P ]` can be rewritten into `{a1}{a2}...{ak}Pk` (see [48]).

By using the Maude 2.0 `search` command, we can find all the possible one-step successors of a process, or all the successors after performing a given action.

$$\text{Weak}\quad \frac{P \xrightarrow{\tau}{}^{*} Q \quad Q \xrightarrow{a} Q' \quad Q' \xrightarrow{\tau}{}^{*} P'}{P \overset{a}{\Longrightarrow} P'}$$

$$\text{Refl*}\quad \frac{}{P \xrightarrow{\tau}{}^{*} P} \qquad\qquad \text{Tran*}\quad \frac{P \xrightarrow{\tau} P' \quad P' \xrightarrow{\tau}{}^{*} P''}{P \xrightarrow{\tau}{}^{*} P''}$$

Figure 24: CCS weak transition.

```
Maude> search 'a . 'b . 0 | ~ 'a . 0 =>+ AP:ActProcess .
Solution 1 (state 1)
  AP:ActProcess --> {~ 'a}0 | 'a . 'b . 0
Solution 2 (state 2)
  AP:ActProcess --> {'a}'b . 0 | ~ 'a . 0
Solution 3 (state 3)
  AP:ActProcess --> {tau}0 | 'b . 0
No more solutions.

Maude> search 'a . 'b . 0 + 'c . 0 =>+ {'a}AP:ActProcess .
Solution 1 (state 2)
  AP:ActProcess --> 'b . 0
No more solutions.
```

If we add the following equation to the module `CCS-SEMANTICS`, defining the recursive process `'Proc` in the CCS context, we prove that `'Proc` can perform the trace `'a 'b 'a`:

```
  eq context =  'Proc =def 'a . 'b . 'Proc .

Maude> search [1] [ 'Proc ] =>+ {'a}{'b}{'a}X:Process .
Solution 1 (state 5)
  X:Process --> 'b . 'Proc
```

We have asked Maude to search if there is one (`[1]`) way in which the term `[ 'Proc ]` can be rewritten into the pattern `{'a}{'b}{'a}X:Process`. The `search` command performs a breadth-first search in the conceptual tree of all possible rewrites of term `[ 'Proc ]`, and since there is a solution, it finds it. However, if we asked to search for more solutions, the search would not terminate, although there are no more solutions, because the search tree is infinite.

$$\begin{tikzcd}
 & & \texttt{['Proc]} \arrow[dll] \arrow[dl] \arrow[d] & \cdots & \\
\texttt{\{'a\}'b.'Proc} & \texttt{\{'a\}\{'b\}'Proc} & \texttt{\{'a\}\{'b\}\{'a\}'b.'Proc} & & \cdots
\end{tikzcd}$$

6.3 Extension to weak transition semantics

Another important transition relation defined for CCS, $P \overset{a}{\Longrightarrow} P'$, does not observe τ transitions [52]. It is defined in the first row of Figure 24, where $\xrightarrow{\tau}{}^{*}$ denotes the reflexive, transitive closure of $\xrightarrow{\tau}$, which is also defined as indicated in the second row of Figure 24.

We can also implement this transition relation by means of rewrites; a transition $P \xrightarrow{\tau}{}^{*} P$ will be represented as a rewrite $P \longrightarrow \{\tau\}{*}\ P'$ and a transition $P \overset{a}{\Longrightarrow} P'$ will be represented as a rewrite $P \longrightarrow \{\{a\}\}P'$. We again have to introduce dummy operators to prevent undesired uses of the new rewrite rules in the verification of the rewrite conditions. The proposed implementation is as follows:

```
  sorts Act*Process ObsActProcess .
  op {tau}*_  : Process -> Act*Process [frozen] .
  op {{_}}_ : Act Process -> ObsActProcess [frozen] .
  sort WProcess .
```

$$\begin{array}{ll}
P \models \texttt{tt} & \\
P \models \Phi_1 \wedge \Phi_2 & \text{iff } P \models \Phi_1 \text{ and } P \models \Phi_2 \\
P \models \Phi_1 \vee \Phi_2 & \text{iff } P \models \Phi_1 \text{ or } P \models \Phi_2 \\
P \models [K]\Phi & \text{iff } \forall\, Q \in \{P' \mid P \xrightarrow{a} P' \wedge a \in K\} \,.\, Q \models \Phi \\
P \models \langle K\rangle\Phi & \text{iff } \exists\, Q \in \{P' \mid P \xrightarrow{a} P' \wedge a \in K\} \,.\, Q \models \Phi \\
P \models [\![K]\!]\Phi & \text{iff } \forall\, Q \in \{P' \mid P \overset{a}{\Longrightarrow} P' \wedge a \in K\} \,.\, Q \models \Phi \\
P \models \langle\!\langle K\rangle\!\rangle\Phi & \text{iff } \exists\, Q \in \{P' \mid P \overset{a}{\Longrightarrow} P' \wedge a \in K\} \,.\, Q \models \Phi
\end{array}$$

Figure 25: Modal logic satisfaction relation.

```
  subsorts WProcess < Act*Process ObsActProcess .
  op |_| : Process -> WProcess [frozen] .
  op <_> : Process -> WProcess [frozen] .
  rl  [Refl*] : | P | => {tau}* P .
  crl [Tran*] : | P | => {tau}* R if P => {tau}Q /\ | Q | => {tau}* R .
  crl [Weak] : < P > => {{A}} P' if | P | => {tau}* Q  /\
                                    Q => {A}Q'  /\  | Q' | => {tau}* P' .
```

Notice that both the new semantics operators, `{tau}*_` and `{{_}}_`, as well as the dummy operators, `|_|` and `<_>`, are declared frozen, for the same reasons already explained in Section 6.2.

We can use the `search` command to look for all the weak successors of a given process after performing action `'a`.

```
Maude> search < tau . 'a . tau . 'b . 0 > =>+ {{ 'a }}AP:ActProcess .
Solution 1 (state 2)
  AP:ActProcess --> tau . 'b . 0
Solution 2 (state 3)
  AP:ActProcess --> 'b . 0
No more solutions.
```

6.4 Hennessy-Milner modal logic

We now want to implement the Hennessy-Milner modal logic for describing local capabilities of CCS processes [38, 63]. Formulas are built according to the following grammar:

$$\Phi ::= \texttt{tt} \mid \texttt{ff} \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [K]\Phi \mid \langle K\rangle\Phi \mid [\![K]\!]\Phi \mid \langle\!\langle K\rangle\!\rangle\Phi,$$

where K is a (finite) set of actions. The satisfaction relation describing when a process P satisfies a property Φ, denoted $P \models \Phi$, is inductively defined in Figure 25.

We have that any process satisfies the formula `tt` and none satisfies the formula `ff`. A process P satisfies the formula $\Phi_1 \wedge \Phi_2$ if it satisfies both Φ_1 and Φ_2, and it satisfies the formula $\Phi_1 \vee \Phi_2$ if it satisfies either Φ_1 or Φ_2. A process P satisfies the formula $[K]\Phi$ built with the universal (box) modal operator if all the one-step successors of P after performing an action in the set K satisfy the formula Φ. On the other hand, a process P satisfies the formula $\langle K\rangle\Phi$ built with the existential (diamond) modal operator if at least one of its K successors satisfies Φ.

Since the definition of the satisfaction relation uses the transitions of CCS, we could try to implement it at the same level, with rules like the following ones:

```
  rl [and] : P |= Phi /\ Psi => true if P |= Phi => true /\ P |= Psi => true .
  rl [dia] : P |= < A > Phi => true if P => {A}Q /\ Q |= Phi => true .
```

which implement the behaviour of the conjunction and of the existential modal operator.

These rules are correct, and they exactly represent what the satisfaction relation of the modal logic expresses. For example, the condition of the second rule represents that *there exists* a process

Q such that $P \xrightarrow{a} Q$ and $Q \models \Phi$, which is the definition of the diamond modal operator. That is because the variable Q is (implicitly) existentially quantified in the rule condition. But we find a problem with the definition of the box modal operator, because it uses a universal quantifier over the possible transitions of a process. If we want to work with *all* the possible one-step rewrites of a term, we need to go up to the metalevel. By using the operation metaSearch, we have defined an operation succ that returns all the (metarepresented) successors of a process after performing actions in a given finite set.

The definition of the operation succ in the module SUCC below uses two auxiliary operations. The evaluation of allOneStep(T,N,X) returns all the one-step rewrites of term T (skipping the first N solutions) that match the pattern X by using rules in the module MOD (the metarepresentation of CCS-SEMANTICS denoted by the term ['CCS-SEMANTICS] in the module SUCC shown below). The evaluation of filter(F,TS,AS) returns the metarepresented processes P such that the term F[A,P] is in TS and A is in AS. In order to look for the term A in the term set AS, we compare terms in the module MOD. This is because different metarepresented terms, like ''a.Qid and ''a.Act, can represent the same action in the module CCS-SEMANTICS. The operation filter is used in the definition of succ(T,TS) to remove from all the successors of process T those processes that are reached by performing an action not in the set TS.

Having defined these operations in such a general form, we can implement the operation wsucc that returns all the weak successors with the same operations.

```
fmod SUCC is
  including META-LEVEL .
  op MOD : -> Module .
  eq MOD = ['CCS-SEMANTICS] .
  sort TermSet .
  subsort Term < TermSet .
  op mt : -> TermSet .
  op _U_ : TermSet TermSet -> TermSet [assoc comm id: mt] .
  op _isIn_ : Term TermSet -> Bool .
  op allOneStep : Term MachineInt Term -> TermSet .
  op filter : Qid TermSet TermSet -> TermSet .
  op succ : Term TermSet -> TermSet .
  op wsucc : Term TermSet -> TermSet .
  var M : Module .       var F : Qid .            vars T T' X : Term .
  var N : MachineInt .   vars TS AS : TermSet .
  eq T isIn mt = false .
  eq T isIn (T' U TS) =
     (getTerm(metaReduce(MOD, '_==_[T,T'])) == 'true.Bool) or (T isIn TS) .
  eq filter(F, mt, AS) = mt .
  ceq filter(F, X U TS, AS) =
      (if T isIn AS then T' else mt fi) U filter(F, TS, AS)
      if F[T,T'] := X .
  eq allOneStep(T,N,X) =
     if metaSearch(MOD, T, X, nil, '+, 1, N) == failure then mt
     else  getTerm(metaSearch(MOD, T, X, nil, '+, 1, N)) U
           allOneStep(T, N + 1, X) fi .
  eq succ(T,TS) = filter(('`{_`}_),
                         allOneStep(T, 0, 'AP:ActProcess), TS) .
  eq wsucc(T,TS) = filter(('`{`{_`}`}_),
                          allOneStep('<_>[T], 0, 'OAP:ObsActProcess), TS) .
endfm
```

Using the operations succ and wsucc we have equationally implemented the satisfaction relation of the modal logic. Notice how the semantics for the modal operators is defined by unfolding to a conjunction or disjunction where the successors of the given process are used.

```
fmod MODAL-LOGIC is
  protecting SUCC .
  sort HMFormula .
  ops tt ff : -> HMFormula .
  ops _/\_  _\/_ : HMFormula HMFormula -> HMFormula .
  ops <_>_  [_]_ : TermSet HMFormula -> HMFormula .
  ops <<_>>_ [[_]]_ : TermSet HMFormula -> HMFormula .
  ops forall exists : TermSet HMFormula -> Bool .
  op _|=_ : Term HMFormula -> Bool .
  var P : Term .   vars K PS : TermSet .   vars Phi Psi : HMFormula .
  eq P |= tt           =  true .
  eq P |= ff           =  false .
  eq P |= Phi /\ Psi   =  P |= Phi and P |= Psi .
  eq P |= Phi \/ Psi   =  P |= Phi or  P |= Psi .
  eq P |= [ K ] Phi    =  forall(succ(P, K), Phi) .
  eq P |= < K > Phi    =  exists(succ(P, K), Phi) .
  eq P |= [[ K ]] Phi  =  forall(wsucc(P, K), Phi) .
  eq P |= << K >> Phi  =  exists(wsucc(P, K), Phi) .
  eq forall(mt, Phi)  = true .
  eq forall(P U PS, Phi) = P |= Phi and forall(PS, Phi) .
  eq exists(mt, Phi)  = false .
  eq exists(P U PS, Phi) = P |= Phi or  exists(PS, Phi) .
endfm
```

Using two examples from [63], we show how we can prove in Maude that a modal formula is satisfied by a CCS process. The first example deals with a vending machine `'Ven` defined in a CCS context as follows:

```
  eq context =  ('Ven    =def  '2p . 'VenB  +  '1p . 'VenL)  &
                ('VenB   =def  'big . 'collectB . 'Ven)      &
                ('VenL   =def  'little . 'collectL . 'Ven) .
```

The process `'Ven` may accept, initially, a `2p` or `1p` coin. If a `2p` coin is deposited, the `big` button may be pressed, and a big item can be collected. If a `1p` coin is deposited, the `little` button may be pressed, and a little item can be collected. After an item is collected, the vending machine goes back to the initial state.

One of the properties that the vending machine satisfies is that buttons cannot be pressed initially, that is, before a coin is inserted. We can prove in Maude that `Ven |= [big, little]ff`:

```
Maude> red ''Ven.Qid |= [ ''big.Act + ''little.Act ] ff .
result Bool: true
```

It also satisfies that after a coin is deposited and the corresponding button is pressed, an item (big or little) can be collected.

```
Maude> red ''Ven.Qid |= [ ''1p.Act + ''2p.Act ] [ ''big.Act + ''little.Act ]
                        < ''collectB.Act + ''collectL.Act > tt .
result Bool: true
```

We can also prove that a process does not satisfy a given formula. For example, we can prove that after inserting a coin `1p` in the vending machine, it is not possible to press button `big` and collect a big item.

```
Maude> red ''Ven.Qid |= < ''1p.Act > < ''big.Act > < ''collectB.Act > tt .
result Bool: false
```

The second example deals with a railroad crossing system specified as follows:

```
eq context =  ('Road  =def  'car . 'up . ~ 'ccross . ~ 'down . 'Road)   &
   ('Rail   =def  'train . 'green . ~ 'tcross . ~ 'red . 'Rail)         &
   ('Signal =def  ~ 'green . 'red . 'Signal + ~ 'up . 'down . 'Signal)  &
   ('Crossing =def (('Road | ('Rail | 'Signal))
                    \ 'green \ 'red \ 'up \ 'down ))  .
```

The system consists of three components: `Road`, `Rail`, and `Signal`. Actions `car` and `train` represent the approach of a car and a train, `up` opens the gates for the car, $\overline{\texttt{ccross}}$ is the car crossing, `down` closes the gates, `green` is the receipt of a green signal by the train, $\overline{\texttt{tcross}}$ is the train crossing, and `red` sets the light red.

The process `'Crossing` satisfies that when a car and a train arrive to the crossing, exactly one of them has the possibility to cross it.

```
Maude> red ''Crossing.Qid |= [[ ''car.Act ]] [[ ''train.Act ]]
          ((<< '~_[''ccross.Act] >> tt) \/ (<< '~_[''tcross.Act] >> tt)) .
result Bool: true
Maude> red ''Crossing.Qid |= [[ ''car.Act ]] [[ ''train.Act ]]
          ((<< '~_[''ccross.Act] >> tt) /\ (<< '~_[''tcross.Act] >> tt)) .
result Bool: false
```

7 Full LOTOS

In this section we go one step further in the implementation of structural operational semantics, by presenting a complete tool, implemented all in Maude, where Full LOTOS specifications can be entered and executed.

The formal description technique LOTOS [44] was developed within ISO for the formal specification of open distributed systems. Its behaviour description part is based on process algebras, borrowing ideas from CCS [52] and CSP [41], and the mechanism for defining data types is based on ACT ONE [27]. The union of the behaviour and data type description parts is known as Full LOTOS; we normally use here the term LOTOS to refer to the whole language. LOTOS became an international standard (IS-8807) in 1989; since then LOTOS has been used to describe hundreds of systems, and most of this success is due to the existence of tools where specifications can be executed, compared, and analyzed. A lot of work has been done regarding LOTOS implementations [35, 34, 26, 19, 29].

The standard defines LOTOS semantics by means of labelled transition systems, where each data variable is instantiated with every possible value. That is the reason why most of the tools ignore or restrict the use of data types. Calder and Shankland [10] have defined a *symbolic* semantics for LOTOS which gives meaning to symbolic, or data parameterised processes (see Section 7.1) and avoids infinite branching.

In this section we focus on the use of rewriting logic and Maude to implement, following the transitions as rewrites approach, a complete formal tool based on a symbolic semantics where LOTOS specifications can be executed without having to impose restrictions in their data types. The reflective features of rewriting logic and the good properties of Maude as a metalanguage [14] make it possible to implement the whole tool in the same semantic framework. Specifically, we have obtained an efficient implementation of the operational semantics of the behaviour part of LOTOS, which has been integrated with ACT ONE specifications that are automatically translated to functional modules in Maude, and finally we have built an entire environment with parsing, pretty printing, and input/output processing of LOTOS specifications. Our aim has been to implement a formal tool that can be used by everyone without knowledge of the concrete implementation, but where the semantics representation is at a sufficiently abstract level that it can be understood and modified by anybody familiar with operational semantics.

7.1 LOTOS symbolic semantics

The implementation of the LOTOS symbolic semantics given here is based on the work presented in [10] by Calder and Shankland. A symbolic semantics for LOTOS is given by associating a symbolic transition system with each LOTOS behaviour expression P. Following [37], Calder and Shankland define *symbolic transition systems* (STS) as transition systems which separate the data from process behaviour by making the data symbolic. STS are labelled transition systems with variables, both

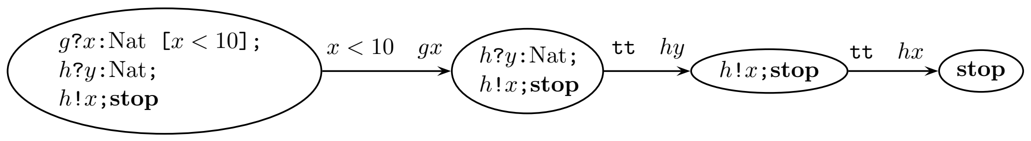

Figure 26: Symbolic transition system.

in states and transitions, and conditions which determine the validity of a transition. A symbolic transition system consists of:

- A (non-empty) set of states. Each state T is associated with a set of free variables, denoted $fv(T)$.
- A distinguished initial state, T_0.
- A set of transitions written as $T \xrightarrow{b \quad \alpha} T'$, where α is an event and b is a Boolean expression, such that $fv(T') \subseteq fv(T) \cup fv(\alpha)$ and $fv(b) \subseteq fv(T) \cup fv(\alpha)$.

In the symbolic semantics, *open* behaviour expressions label states (for example, $h!x$ **; stop**), and transitions offer variables, under some conditions; these conditions determine the set of values which may be substituted for variables.

In [10] the intuition and key features of this semantics are presented, together with axioms and inference rules for each LOTOS operator. We will present some of them, together with their representation in Maude, in Section 7.2.2. Figure 26 shows an example of STS.

As we will see in Section 7.2.3, the obtained representation is itself executable, although without values, apart from the predefined Booleans. The LOTOS symbolic semantics is parameterized over the set of *values* and *data expressions*. Thus, if we want to build a usable formal tool, we also need to handle data types, specified using ACT ONE [27].

Instead of defining a data type for representing ACT ONE modules in Maude and operations to represent the reduction process in ACT ONE, we have implemented an automatic translation from ACT ONE modules into functional modules in Maude. We are then able to use Maude's high-performance reduction engine in a conservative way. We present in Section 7.3 this translation and then show how the modules are extended to be used by the semantics.

In Section 7.4 we show how the semantics implementation and the ACT ONE modules translation are integrated to build an entire environment for our formal tool, where LOTOS specifications with complete freedom in their data types and (possibly recursive) process definitions, can be entered and executed by means of a user interface that completely hides the concrete implementation details.

7.2 LOTOS symbolic semantics in Maude

In order to implement the LOTOS symbolic semantics in Maude following the transitions as rewrites approach, we interpret a LOTOS transition $T \xrightarrow{b \quad \alpha} T'$ as a rewriting logic rewrite $T \longrightarrow \{b\}\{\alpha\}T'$. Since the rewriting logic arrow has no labels, we write them as part of the righthand side term, as we did in the CCS case (see Section 6.2).

In the following, the Maude modules will be abbreviated due to space reasons; the complete code can be found in [69].

7.2.1 LOTOS syntax

There are two different types of syntax: the concrete syntax used by the specifier (see [69], for full details), and the abstract syntax used by the semantic definition and implementation and introduced in this section. It is defined in the Maude functional module `LOTOS-SYNTAX`, which includes `DATAEXP-SYNTAX`. We use the predefined quoted identifiers to build LOTOS variable, sort, gate, and process identifiers. Booleans are the only predefined data type. LOTOS syntax is extended in a user-definable way when ACT ONE data types specifications are used. Values of these data types will extend the type `DataExp` below. We shall see how this is done in Section 7.3.

```
fmod DATAEXP-SYNTAX is
  protecting QID .
  sort VarId .
  op V : Qid -> VarId .
  sort DataExp .
  subsort VarId < DataExp . *** A LOTOS variable is a data expression.
  subsort Bool < DataExp .  *** Booleans are a predefined data type.
endfm

fmod LOTOS-SYNTAX is
  protecting DATAEXP-SYNTAX .
  sorts SortId GateId ProcId .
  op S : Qid -> SortId .  op G : Qid -> GateId .  op P : Qid -> ProcId .
  sort BehExp .
  op stop : -> BehExp .
  op exit(_) : ExitParam -> BehExp .
  op _;_ : Action BehExp -> BehExp [frozen prec 35] .
  op _[]_ : BehExp BehExp -> BehExp [frozen prec 40] .
  op _|[_]|_ : BehExp GateIdList BehExp -> BehExp [frozen prec 40] .
  op hide_in_ : GateIdList BehExp -> BehExp [frozen prec 40] .
  [...]
endfm
```

7.2.2 LOTOS symbolic semantics implementation

First, we define `Contexts`, which are used to keep the definitions of processes introduced in a LOTOS specification. In order to execute a process instantiation, the process definition has to be looked for in the context. The actual context is built when the LOTOS specification is entered to the tool (we will see how this is done in Section 7.4.2). In the semantics, a constant `context` is assumed, representing the collection of process definitions. We could say that the semantics is parameterized over this constant, that will be instantiated when a concrete specification is used.

```
fmod CONTEXT is
  protecting LOTOS-SYNTAX .
  sort Context .
  op context : -> Context .
  [...]
endfm
```

Now, we can implement the LOTOS symbolic semantics. First we show some operations used in the semantic definition that present problems when implementing them, and how we have solved these problems in Maude.

In the semantics, a set **new-var** of fresh variable names is assumed. As mentioned in [10], strictly speaking, any reference to this set requires a context, i.e. the variable names occurring so far. Instead of complicating the implementation with this other context, we have preferred to use a predefined Maude utility imported from module `ORACLE`, where a constant `NEWQ` is defined. Each time `NEWQ` is rewritten, it is rewritten to a different quoted identifier. With the following definition, we have the set of fresh variable names.

```
  op new-var : -> VarId .
  eq new-var = V(NEWQ) .
```

A (data) substitution is written as $[z/x]$, where z is substituted for x. It seems to be easy to implement equationally, and we present below some equations showing how the substitution operation distributes over the syntax of behaviour expressions. However, if we want to allow user-definable data expressions by means of an ACT ONE specification, we cannot completely define this operation now, because we do not know at this point the syntax of data expressions. We will describe in Section 7.3.1 how the module containing the new syntax is automatically extended to define this operation on new data expressions.

```
op _[_/_] : BehExp DataExp VarId -> BehExp .
op _[_/_] : DataExp DataExp VarId -> DataExp .
vars E E' E1 : DataExp .
var g : GateId .  var O : Offer .      var SP : SelecPred .
var x : VarId .   var b : TransCond .  var GIL : GateIdList .
var S : SortId .  var a : Event .      vars P P' P1 : BehaviourExp .
eq stop [E' / E] = stop .
eq g ! E1 ; P [E' / E] = g ! (E1[E' / E]) ; (P[E' / E]) .
eq P1 [] P2 [E' / E] = (P1[E' / E]) [] (P2[E' / E]) .
[...]
```

An operation *vars*, used to obtain the variables occurring in a behaviour expression, gives rise to the same problem, that is, we cannot define it completely at this level since data expressions syntax is user-definable. We will see in Section 7.3.1 how it is extended automatically for new data expressions.

```
sort VarSet .  subsort VarId < VarSet .
op mt : -> VarSet .
op _U_ : VarSet VarSet -> VarSet [assoc comm id: mt] .
eq x U x = x . *** idempotency
op vars : BehExp -> VarSet .
op vars : DataExp -> VarSet .
eq vars(stop) = mt .
eq vars(g ? x : S ; P) = x U vars(P) .
eq vars(P1 [] P2) = vars(P1) U vars(P2) .
eq vars(x) = x .
[...]
```

As mentioned above, a transition $T \xrightarrow{b \quad \alpha} T'$, where T and T' are behaviour expressions, b is a transition condition, and α is an event, will be represented as a rewrite $T \longrightarrow \{b\}\{\alpha\}T'$, where the righthand side term is of sort `TCondEventBehExp`.

```
sort TCondEventBehExp .
subsort BehExp < TCondEventBehExp .
op {_}{_}_ : TransCond Event BehExp -> TCondEventBehExp [frozen] .
```

As in Section 6.2, rewrite rules will be sort-increasing, since they rewrite terms of sort `BehExp` to terms of its supersort `TCondEventBehExp`. This avoids the appearance of bad-formed terms in which subterms of a LOTOS operator are not behaviour expressions.

Moreover, the operator `{_}{_}_` used to build the values in the righthand side of the rewrite rules representing the semantic rules has been declared frozen. The reason is to avoid undesired rewrites which can lead to infinite searches. In this way, with the command `search` or the metalevel operation `metaSearch` we can obtain the one-step successors of a behaviour expression.

The LOTOS symbolic semantics consists of 28 rules. We present some of those rules and their representation as rewrite rules in Figure 27; the complete set of rules can be found in [69]. We also show the inference rules to ease the comparison between the mathematical and Maude representations. The inference rules are not exactly the ones presented in [10], because we have generalized them to allow multiple event offers at an action.

The rules for the prefix operator show how axioms are represented as rewrite rules without conditions. The choice range rule shows how non-deterministic choice can be made by using rewrite rules. The hide rules show how side conditions in the inference rules are added as conditions in the rewrite

prefix axioms

$$a; P \xrightarrow{\texttt{tt} \quad a} P$$

$$g\ d_1 \dots d_n; P \xrightarrow{\texttt{tt} \quad gE'_1 \dots E'_n} P$$

$$g\ d_1 \dots d_n[SP]; P \xrightarrow{SP \quad gE'_1 \dots E'_n} P$$

where $E'_i = \begin{cases} E_i & \text{if} \quad d_i \ = !E_i \\ x_i & \text{if} \quad d_i \ = ?x_i : S_i \end{cases}$

```
rl A ; P => {true}{A}P .
rl g O ; P => {true}{g eOff(O)}P .
rl g O [SP] ; P => {SP}{g eOff(O)}P .

op eOff : Offer -> EOffer .
eq eOff(! E) = E .
eq eOff(? x : S) = x .
eq eOff(O O') = eOff(O) eOff(O') .
```

choice range rule

$$\frac{P[g_i/g] \xrightarrow{b \quad \alpha} P'}{\textbf{choice}\ g\ \textbf{in}\ [g_1, \dots, g_n]\ [\,]\ P \xrightarrow{b \quad \alpha} P'}$$

for each $g_i \in \{g_1, \dots, g_n\}$

```
crl choice g in [GIL][] P => {b}{a}P'
 if select(GIL) => gi  /\  P[gi / g] => {b}{a}P' .
sort GateId? . subsort GateId < GateId? .
op select : GateIdList -> GateId? .
rl select(g) => g .
rl select(g, GIL) => g .
rl select(g, GIL) => select(GIL) .
```

hide rules

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\textbf{hide}\ g_1, \dots, g_n\ \textbf{in}\ P \xrightarrow{b \quad \texttt{i}} \textbf{hide}\ g_1, \dots, g_n\ \textbf{in}\ P'} \quad \textbf{name}(\alpha) \in \{g_1, \dots, g_n\}$$

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\textbf{hide}\ g_1, \dots, g_n\ \textbf{in}\ P \xrightarrow{b \quad \alpha} \textbf{hide}\ g_1, \dots, g_n\ \textbf{in}\ P'} \quad \textbf{name}(\alpha) \notin \{g_1, \dots, g_n\}$$

```
crl hide GIL in P => {b}{i}hide GIL in P'
 if P => {b}{a}P'  /\  (name(a) in GIL) .
crl hide GIL in P => {b}{a}hide GIL in P'
 if P => {b}{a}P'  /\  not(name(a) in GIL) .
```

general parallelism rule (not synchronising)

$$\frac{P_1 \xrightarrow{b \quad \alpha} P'_1}{P_1 |[g_1, \dots, g_n]| P_2 \xrightarrow{b\sigma \quad \alpha\sigma} P'_1\sigma |[g_1, \dots, g_n]| P_2} \quad \textbf{name}(\alpha) \notin \{g_1, \dots, g_n, \delta\}$$

where $\alpha = gE_1 \dots E_n$, $\sigma = \sigma_1 \dots \sigma_n$, $dom(\sigma_i)$ are disjoint, and
$\sigma_i = \begin{cases} [z_i/x_i] & \text{if } E_i = x_i,\ x_i \in vars(P_2) \text{ and } z_i \in \textbf{new-var}. \\ [\,] & \text{otherwise} \end{cases}$

```
crl P1 |[GIL]| P2 => {b sp(a,vars(P2))}{a sp(a,vars(P2))}
                      ((P' sp(a,vars(P2))) |[GIL]| P2)
 if P1 => {b}{a}P' /\ not(name(a) in (GIL, delta)) .
```

Figure 27: Some semantics rules and their implementation in Maude.

rules. Finally, the general parallelism rule shows how external definitions can be used, as the one defining the substitution (not shown in Figure 27).

After having implemented all the semantics rules for behaviour expressions, we have the following conservativity result: Given a LOTOS behaviour expression P, there are a transition condition b, an event a, and a behaviour expression P' such that

$$P \xrightarrow{b \quad a} P'$$

if and only if `P` can be rewritten `{b}{a}P'` using the presented rules.

In [10], the concept of a *term* is also defined, consisting of an STS paired with a substitution. Transitions between terms are then defined. We have also implemented these transitions in a way similar to the implementation of transitions for behaviour expressions (see [69]).

7.2.3 Execution example

By using the Maude `search` command, we can find all the possible transitions of a behaviour expression.

```
Maude> search
  G('g) ; G('h) ; stop
|[ G('g) ]|
  (G('a) ; stop [] G('g) ; stop) =>+ X:TCondEventBehExp .
Solution 1 (state 1)
X:TCondEventBehExp --> {true}{G('a)}G('g) ; G('h) ; stop |[G('g)]| stop
Solution 2 (state 2)
X:TCondEventBehExp --> {true}{G('g)}G('h) ; stop |[G('g)]| stop
No more solutions.

Maude> search G('h) ; stop |[G('g)]| stop =>+ X:TCondEventBehExp .
Solution 1 (state 1)
X:TCondEventBehExp --> {true}{G('h)}stop |[G('g)]| stop
No more solutions.
```

But we have to write behaviour expressions using the abstract syntax (like the gate identifier `G('h)`) and we cannot use data expressions, apart from the predefined Booleans, because we have not introduced yet any ACT ONE specification. These specifications are part of a Full LOTOS specification, and therefore are user-definable. We will see in the following sections how to give semantics to ACT ONE specifications and how they can be integrated with the previous LOTOS semantics implementation.

7.3 ACT ONE modules translation

We want to be able to introduce into our tool user-defined ACT ONE specifications, which will then be translated internally to Maude functional modules.

We first have to define ACT ONE's syntax. In Maude, the *syntax definition* for a language $\mathcal{L}$ is accomplished by defining a data type $\texttt{Grammar}_{\mathcal{L}}$; this can be done with very flexible user-definable *mixfix* syntax, that can mirror the concrete syntax of $\mathcal{L}$. Particularities at the lexical level of $\mathcal{L}$ can be accommodated by user-definable *bubble sorts*, that tailor the adequate notions of token and identifier to the language in question. Bubbles correspond to pieces of a module in a language that can only be parsed once the grammar introduced by the signature of the module is available [14]. This is specially important when $\mathcal{L}$ has user-definable syntax, as it is our case with ACT ONE. The grammar of ACT ONE is defined in a module `ACTONE-GRAMMAR` that can be found in [69].

The idea is that the syntax of a language that allows modules including syntactic characteristics defined by the user can be seen in a natural way as a syntax with two different levels: one that we can call the *top level* syntax of the language, and another one user-definable that is introduced in each module. Bubble sorts allow us to reflect this duplicity of levels. In order to illustrate this concept, let us consider the following ACT ONE specification defining natural numbers modulo 3:

```
type NAT3 is
  sorts Nat3
  opns
    0 : -> Nat3
    s_ : Nat3 -> Nat3
  eqns
    ofsort Nat
      s s s 0 = 0 ;
endtype
```

The boxed character sequences are not part of the top-level syntax of ACT ONE. In fact, they can only be parsed with the grammar associated to the signature in the specification `NAT3`.

After having defined the module with ACT ONE syntax, we can use the metalevel operation `metaParse`, which receives as arguments the representation of a module M and the representation of a list of tokens, and returns the metarepresentation of the parsed term (a parse tree that may have bubbles) of that list of tokens for the signature of M.

The next step consists in defining an operation `translate` that receives the parsed term and returns a functional module with the same semantics as the introduced ACT ONE specification. The syntactic analysis of possible bubbles is also done in this second step.

$$\texttt{QidList} \xrightarrow{\texttt{metaParse}} \texttt{Grammar}_{\text{ACT ONE}} \xrightarrow{\texttt{translate}} \texttt{FModule}$$

Notice that we start with a `QidList` (a list of quoted identifiers), that is obtained from the user input (see Section 7.4.3).

With our translation we achieve the following result: given an ACT ONE specification SP, and terms t and t' in SP, we have

$$SP \models t \equiv t' \iff M \models t_M \equiv t'_M$$

where $M = \texttt{translate}(\texttt{metaParse}(\texttt{ACTONE-GRAMMAR},SP))$, and t_M and t'_M are the representations of t and t' in M.

Before presenting in more detail how the translation is implemented, let us see how these two steps are performed with the previous example module `NAT3`. If we execute the operation `metaParse` with the metarepresented module `ACTONE-GRAMMAR` (which contains the top level syntax of ACT ONE) and the following list of quoted identifiers

```
'type 'NAT3 'is
  'sorts 'Nat3
  'opns
    '0 ': '-> 'Nat3
    's_ ': 'Nat3 '-> 'Nat3
  'eqns
    'ofsort 'Nat
      's 's 's '0 '= '0 ';
'endtype
```

we obtain the next metarepresented term, that includes metarepresented tokens and bubbles:

```
'type_is_endtype[
  'token[''NAT3.Qid],
  '__['sorts_['token[''Nat3.Qid]],
  '__['opns_[
      '__['_:`->_['token[''0.Qid],'token[''Nat3.Qid]],
          '_:_->_['token[''s_.Qid],'token[''Nat3.Qid],'token[''Nat3.Qid]]]],
      'eqns_[
         'ofsort__['token[''Nat.Qid],
         '_=_;['bubble['__[''s.Qid,''s.Qid,''s.Qid,''0.Qid]],'bubble[''0.Qid]]
                  ]
            ]
```

```
    ]
    ]
              ]
```

Tokens and bubbles have as arguments metarepresented lists of quoted identifiers, that is, values of sort `QidList` metarepresented. These values have to be parsed again (going down one level in the representation) with the user-defined syntax (given in the `opns` part of the ACT ONE specification).

If we now execute the `translateType` operation commented below, we obtain the following metarepresented Maude functional module, of sort `FModule`:

```
fmod 'NAT3 is
  including 'DATAEXP-SYNTAX .
  sorts 'Nat3 .
  subsort 'VarId < 'Nat3 .
  subsort 'Nat3 < 'DataExp .
  op '0 : nil -> 'Nat3 [none] .
  op 's_ : 'Nat3 -> 'Nat3 [none] .
  none
  eq 's_['s_['s_['0.Nat3]]] = '0.Nat3 .
endfm
```

This functional module is the translation into Maude of the `NAT3` specification in ACT ONE that we have seen at the beginning of this section.

The translation is performed by the following operations defined at the metalevel:

```
  op translateType : Term -> FModule .
  op translateType : Term FModule FModule -> FModule .
  op translateDeclList : Term FModule FModule -> FModule .
```

The first operation is the main one. It receives as argument the term returned by `metaParse` and it returns its translation as a functional module. To do that it uses the generalized operation `translateType` with three arguments: the ACT ONE specification not yet translated, the Maude module with the already made translation, and the Maude module with (only) the signature in the translated part.

```
  vars T T' T'' : Term .
  vars M M' : Module .
  eq translateType(T) =
     translateType(T, addImportList(including 'DATAEXP-SYNTAX ., emptyFModule),
                   emptyFModule) .
  eq translateType('type_is_endtype['token[T'],T''], M, M') =
     translateDeclList(T'', M,
       addDecls(M',extractSignature('type_is_endtype['token[T'],T'']))) .
```

The operation `translateDeclList` goes through the list of declarations inside a data type specification, and it adds to its second argument the translation of each element.

For example, when an ACT ONE sort declaration for sort `T` is found, it is not only translated into a Maude sort declaration for sort `T`, but we also have to declare the type `T` as a subsort of the sort `DataExp` (since values of the declared type could be used in a behaviour expression to be communicated) and the sort of LOTOS variables `VarId` has to be declared as a subsort of the type `T` (since LOTOS variables could be used to build data expressions of this type). This is done in this way because we want to integrate ACT ONE modules with LOTOS specifications, but the translation is useful by itself, since it provides us with a tool in Maude where ACT ONE specifications can be entered and executed.

```
  eq translateDeclList('sorts_['token[T]], M, M') =
     addSubsortDeclSet(subsort downQid(T) < 'DataExp .,
       addSubsortDeclSet(subsort 'VarId < downQid(T) .,
         addSortSet(downQid(T), M))) .
```

```
type Naturals is
  sorts Nat
  opns
    0 : -> Nat
    s : Nat -> Nat
    _+_ : Nat, Nat -> Nat
  eqns
    forall x, y : Nat
    ofsort Nat
      0 + x = x ;
      s(x) + y = s(x + y) ;
endtype
```

$\Longrightarrow$

```
fmod Naturals is
  including DATAEXP-SYNTAX .
  sorts Nat .
  subsort VarId < Nat .
  subsort Nat < DataExp .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  eq 0 + x:Nat = x:Nat .
  eq s(x:Nat) + y:Nat =
        s(x:Nat + y:Nat) .
endfm
```

Figure 28: ACT ONE specification translation.

We show another translation example in Figure 28. The ACT ONE specification on the left is translated into the functional module on the right.

7.3.1 Module extensions

In Section 7.2.2 we saw that the operation that performs the syntactic substitution and the operation that extracts the variables occurring in a behaviour expression were not completely defined. The reason why we cannot define them completely when defining the semantics is the same in both cases: the presence of data expressions with user-definable syntax, and thus unknown at that moment.

Now that we know the ACT ONE specification and we have translated it to a functional module, we can define these operations on data expressions using the new syntax. Due to the metaprogramming features of Maude, we can do it automatically. We have defined operations that take a module M and return the same module M but where equations defining the substitution and extraction of variables over expressions built using the signature in M have been added.

For example, if the operation `addOpervars` is applied to the module `Naturals` above, it adds the following equations:

```
eq vars(0) = mt .
eq vars(s(v1:Nat)) = vars(v1:Nat) .
eq vars(v1:Nat + v2:Nat) = vars(v1:Nat) U vars(v2:Nat) .
```

Notice that in principle this is not the most natural way of defining this operation over `Nat` terms, because the only constructors of sort `Nat` are `0` and `s`, and hence one could think that the first two equations would be enough. However, here we are defining the operation over expressions that can contain LOTOS variables, so the third equation is also needed.

We next explain how the operation `addOpervars` is implemented. Its argument is a module M corresponding to the translation of an ACT ONE specification. So the operations declared in M can be used to build LOTOS expressions of a certain sort. The operation `addOpervars` goes through the list of operator declarations, and for each of them it adds an equation defining how variables are extracted from terms whose top operator is that one. The module `UNIT` used below includes operations for building metarepresented modules from their components: sort declarations, operations, equations, rules, etc.

```
fmod MODULE-EXTENSIONS is
  protecting UNIT .
  op addOpervars : Module -> Module .
  op addOpervars : OpDeclSet Module -> Module .
  op addOpervars : Qid TypeList Qid Module -> Module .
  op buildArgs : TypeList MachineInt -> TermList .
  op buildArgs2 : Qid TermList -> TermList .
  var M : Module .       vars OP S A A' : Qid .  var ARGS : TypeList .
  var T : Term .         var TL : TermList .     var AttS : AttrSet .
  var ODS : OpDeclSet .  var N : MachineInt .
  eq addOpervars(M) = addOpervars(opDeclSet(M), M) .
  eq addOpervars(none, M) = M .
  eq addOpervars(op OP : ARGS -> S [AttS] . ODS, M) =
      addOpervars(ODS, addOpervars(OP, ARGS, S, M)) .
  eq addOpervars(OP, nil, S, M) =
      addEquationSet(eq 'vars[conc(OP,conc('.,S))] = 'mt.VarSet ., M) .
  eq addOpervars(OP, A ARGS, S, M) =
      addEquationSet(eq 'vars[OP[buildArgs(A ARGS, 1)]] =
         if ARGS == nil then 'vars[buildArgs(A ARGS, 1)]
         else '_U_[buildArgs2('vars, buildArgs(A ARGS, 1))] fi ., M) .
  eq buildArgs(A, N) = conc(conc(index('v,N),':), A) .
  eq buildArgs(A A' ARGS, N) = buildArgs(A, N), buildArgs(A' ARGS, N + 1) .
  eq buildArgs2(OP, T) = OP[T] .
  eq buildArgs2(OP, (T,TL)) = OP[T], buildArgs2(OP,TL) .
endfm
```

The equations added by the operation `addOpervars` together with the equation

```
  eq vars(x) = x .
```

that we saw in Section 7.2.2 define how to extract variables from LOTOS data expressions built with user-defined syntax.

7.4 Building the LOTOS user interface

We want to implement a formal tool where complete LOTOS specifications (with ACT ONE data type specifications, a main behaviour expression, and process definitions) are entered and executed. In order to *execute* or *simulate* the specification, we want to be able to traverse the symbolic transition system generated for the main behaviour expression by using the symbolic semantics instantiated with the data types given in ACT ONE and the given process definitions. We present here the main ideas used in our implementation; full details can be found in [69].

The following module defines the commands of our tool.

```
fmod LOTOS-TOOL-SIGN is  protecting LOTOS-SIGN .
  sort LotosCommand .
  op show process . : -> LotosCommand .
  op show transitions . : -> LotosCommand .
  op show transitions of_. : BehExp -> LotosCommand .
  op cont_. : MachineInt -> LotosCommand .
  op cont . :  -> LotosCommand .
  op show state . : -> LotosCommand .
endfm
```

The first command is used to show the current process. The second and third commands are used to show the possible transitions (defined by the symbolic semantics) of the current or explicitly given process, that is, they start the execution of a process. The fourth command is used to continue the execution with one of the possible transitions, the one indicated in the argument of the command. Command `cont` is a shorthand for `cont 1`. The sixth command is used to show the current *state* of execution, that is, the current condition, trace, and possible next transitions.

7.4.1 LOTOS input processing

When LOTOS behaviour expressions are introduced, either as part of a whole specification or in a tool command, they have to be transformed into elements of the data type `BehaviourExp` in the module `LOTOS-SYNTAX` (Section 7.2.1). The parse tree returned by `metaParse` with module `LOTOS-GRAMMAR` may have bubbles (where data expressions may appear) that have to be parsed again using the user-defined syntax. This syntax is obtained by translating the types defined in ACT ONE into functional modules, as explained above. Moreover, the behaviour itself can define new syntax, since it can declare new LOTOS variables by means of `?` offers, and these variables may appear in expressions. For example, when processing the behaviour expression

```
g ? x : Nat ; h ! s(x) + s(0) ; stop
```

the data expression `s(x) + s(0)` should be parsed using the fact that `x` is a variable of sort `Nat`.

We use the operation `parseProcess` to perform this translation. It takes as arguments the term returned by `metaParse` (representing a behaviour expression), the metarepresented module with the data types syntax (obtained from the ACT ONE specification), and the set of free variables that may appear in the behaviour expression. It returns a behaviour expression without bubbles. It uses the operation `parseAction` that, besides the term metarepresenting the given action (without bubbles), returns the variables declared in the action (if any).

The operation `parseDataExp` takes an expression with bubbles, a module with the syntax with which the expression has to be parsed, and a set of LOTOS variables which may appear in the expression (that is, the expression was found in the *scope* of these variables). In order to correctly parse the expression with bubbles, information about the variables has to be included in the expression as Maude variables. The resulting term may have Maude variables, that have to be transformed into LOTOS variables (which have the form `V(Q)`, where `Q` is a quoted identifier).

Finally, the operation `parseProcDeclList` is used to build a metarepresented context that includes the definitions of the processes declared in a specification.

7.4.2 Tool state handling

In our tool, the persistent state of the system is given by a single object which maintains the tool state. This object has the following attributes:

- `semantics`, to keep the actual module where behaviour expressions can be executed, that is, the module `LOTOS-SEMANTICS` in Section 7.2.2 extended with the syntax and semantics for new data expressions;
- `lotosProcess`, to keep the behaviour expression that labels the node in the symbolic transition system that has been reached during the execution;
- `transitions`, to keep the set of possible transitions from `lotosProcess`;
- `trace`, to keep the sequence of events performed in the path from the root of the STS to the current node;
- `condition`, to keep the conjunction of transition conditions in that path; and
- `input` and `output`, to handle the communication with the user.

We declare the following class by using the notation for classes in object-oriented modules [16]:

```
class ToolState | semantics : Module, lotosProcess : Term,
                  transitions : TermSeq, trace : Term, condition : Term,
                  input : QidList, output : QidList .
```

Then we describe by means of rewrite rules the behaviour of the tool when a LOTOS specification or the different commands are entered into the system. For example, there is a rule which processes a LOTOS specification entered into the system. We allow LOTOS specifications with four arguments: the name of the specification, an ACT ONE specification defining the data types to be used, the main behaviour expression, and a list of process definitions (either the ACT ONE specification or the list of processes can be empty). No local declarations are allowed. When a specification is entered, the `semantics` attribute is set to a new module built as follows: first, the ACT ONE part of the specification is translated to a functional module; then, equations defining the extraction of variables and substitution are added (as explained in Section 7.3.1); the resulting module is joined with the metarepresentation of module `LOTOS-SEMANTICS`; and, finally, an equation defining the constant `context` (Section 7.2.2) with the definitions of processes given in the specification is added. The `lotosProcess` attribute is also updated to the behaviour expression in the introduced specification (after having converted it to a term of sort `BehExp`), and the rest of attributes are initialized.

```
rl [spec] :
    < O : X@ToolState |
      input : ('specification__behaviour_where_endspec['token[T],
                                                  T',T'',T''']),
      output : nil,
      semantics : SemM, lotosProcess : T1,
      transitions : TS,
      trace : T3,
      condition : T4, Atts >
   => < O : X@ToolState | input : nilTermList,
         output : ('\n 'Introduced 'specification getName(T) '\n),
         semantics : addEquationSet(eq 'context.Context =
                           parseProcDeclList(T''',
                             addDecls(translateType(T'), SYN)) .,
                       addDecls(SEM, addOperSubs(
                                   addOpervars(translateType(T'))))),
         lotosProcess : parseProcess(T'',
                          addDecls(translateType(T'), SYN),mt),
         transitions : mt,
         trace : 'nil.Trace,
         condition : 'true.Bool, Atts > .
```

Tool commands are handled by rules as well. For example, there is a rule that handles the `show transitions` command. It modifies the `transitions` attribute by using an operation which receives a module with the semantics implementation (extended with the syntax and semantics of data expressions) and a term t representing a behaviour expression, and returns the sequence of terms representing the possible transitions of t. It uses the operation `metaSearch` that represents at the metalevel the `search` command used in Section 7.2.3.

7.4.3 The LOTOS tool environment

Input/output of specifications and of commands is accomplished by the predefined module `LOOP-MODE` [16], that provides a generic read-eval-print loop. This module has an operator `[_,_,_]` that can be seen as a persistent object with an input and output channel (the first and third arguments, respectively), and a state (given by its second argument). We have complete flexibility for defining this state. In our tool we use an object of the `ToolState` class. When something is written in the Maude prompt enclosed in parentheses it is placed in the first slot of the loop object, as a list of quoted identifiers. Then it is parsed by using the adequate grammar, and the parsed term is put in the `input` attribute of the tool state object. Finally, the rules describing the tool state handling process it. The output is handled in the reverse way, that is, the list of quoted identifiers placed in the third slot of the loop is printed on the terminal.

7.5 An execution example

We give an example of an interaction with the LOTOS tool. Although we use here a very simple example, we have used the tool to execute larger examples [69], including the Alternating Bit Protocol and the Sliding Window Protocol (with more than 550 lines of code) [67]. Our tool has proved to be quite practical, giving the answer to the entered commands in a few milliseconds.

```
Maude> (specification SPEC
type Naturals is
  [as shown above]
endtype
behaviour
 h ! 0 ; stop [] (  g ! (s(0)) ; stop
                  |[ g ]|
                    g ? x : Nat ; h ! (x + s(0)) ; stop )
endspec)
```

```
Maude> (show transitions .)
Trace : nil
Condition : true
TRANSITIONS :
1.  {true}{h 0}stop
2.  {x = s(0)}{g s(0)}stop |[g]| h ! s(s(0)); stop

Maude> (cont 2 .)
Trace : g s(0)
Condition : x = s(0)
TRANSITIONS :
1.  {true}{h s(s(0))}stop |[g]| stop

Maude> (cont .)
Trace :(g s(0))(h s(s(0)))
Condition : x = s(0)
No more transitions .
```

7.6 Comparison with other LOTOS tools

The Concurrency Workbench of the New Century (CWB-NC) [19] is an automatic verification tool where systems in several specification languages can be executed and analyzed. Regarding LOTOS, CWB-NC accepts Basic LOTOS, because it does not support value-passing process algebras. The design of the system exploits the language-independence of its analysis routines by localizing language-specific procedures, which enables users to change the system description language by using the Process Algebra Compiler, that translates the operational semantics definitions into SML code. We have followed a similar approach, although we have tried to keep the semantics representation at a very abstract level, without losing executability. We have also implemented the semantics of the Hennessy-Milner modal logic for CCS (Section 6.4) and the subset of FULL [9] corresponding to this logic for LOTOS. Both implementations follow the same idea, using an operation to calculate the one-step successors of a process, which in turn uses the operational semantics definitions. Thus, the implementation of the formal analysis algorithm, that is, the representation in Maude of the modal logic semantics, is the same in both cases, resulting in similar achievements as the CWB-NC on keeping separated the language-specific features from the general ones.

The Caesar/Aldebaran Development Package (CADP) [29] is a toolbox for protocol engineering, with several functionalities, from interactive simulation (as we do in our tool) to formal verification. In order to support different specification languages, CADP uses low-level intermediate representations, which forces the implementer of a new semantics to write compilers that generate these representations. CADP has already been used to implement FULL [8], although with the severe restrictions to finite types and to the standard semantics of LOTOS instead of the symbolic one in which FULL is based.

8 Transitions as judgements

As we mentioned in the introduction, a very general possibility to represent in rewriting logic an operational semantics consists in mapping an inference rule of the form

$$\frac{S_1 \ldots S_n}{S_0}$$

into a rewrite rule of the form $S_0 \longrightarrow S_1 \ldots S_n$ that rewrites multisets of judgements going from the conclusion to the premises, so that rewriting with these rewrite rules corresponds to searching for a proof in a bottom-up way. We summarize here the main ideas used in this approach, where transitions become judgements and inference rules become rewrites. To illustrate the ideas we use the evaluation semantics of *Fpl* presented in Section 3.2. An implementation of the CCS operational semantics and the Hennessy-Milner modal logic using this approach can be found in [71].

We can use directly the modules `FPL-SYNTAX` (with the syntax of *Fpl*), `AP` (with the definition of the application operation *Ap* used by the semantics), and `ENV` (with the definition of the environments for variables) given in Section 3.1, since they are independent of the operational semantics representation.

In order to represent the semantic rules, a judgement $D, \rho \vdash e \Longrightarrow_A v$ is represented by a *term* `D,rho |- e ==>A v` of sort `Judgement`, built by means of the following operator (it is important not to confuse the arrow which is part of the operator with the arrow in a rewrite rule):

```
sort Judgement .
op _,_|-_==>A_ : Dec ENV Exp Num -> Judgement [prec 50] .
```

In general, a semantic rule has a conclusion and a set of premises, each one represented by means of a judgement. Thus we need a data type for representing sets of judgements:

```
sort JudgementSet .
subsort Judgement < JudgementSet .
op emptyJS : -> JudgementSet .
op __ : JudgementSet JudgementSet -> JudgementSet [assoc comm id: emptyJS prec 60] .
```

The union constructor is written with empty syntax (`__`), and declared associative (`assoc`), commutative (`comm`), and with the empty set as identity element (`id: emptyJS`). Matching and rewriting take place *modulo* such properties, allowing in this way a more abstract treatment of syntax.

A semantic rule is implemented as a rewrite rule where the singleton set consisting of the judgement representing the conclusion is rewritten to the set consisting of the judgements representing the premises. Axiom schemas (semantic rules without premises), like CR or VarR in Figure 2, are represented by means of rewrite rules that rewrite the representation of the conclusion to the empty set of judgements. If the semantic rule has a side condition, it is represented as a Boolean condition in a conditional rewrite rule. Next we show some examples:[1]

```
rl [CR] :  D,rho |- n ==>A n
       => ------------------
             emptyJS .
crl [VarR] :  D,rho |- x ==>A v
          => ------------------
                 emptyJS
          if v == rho(x) .
crl [OpR] :  D,rho |- e op e' ==>A v''
         => -------------------------
              D,rho |- e ==>A v
              D,rho |- e' ==>A v'
         if v'' == Ap(op, v, v') .
rl [IfR1] :  D,rho |- If be Then e Else e' ==>A v
         => ------------------------------------
              D,rho |- be ==>B T
              D,rho |- e  ==>A v .
rl [IfR2] :  D,rho |- If be Then e Else e' ==>A v'
         => -------------------------------------
              D,rho |- be ==>B F
              D,rho |- e'  ==>A v' .
```

[1]By using the fact that text beginning with `---` is a comment in Maude, the rules are displayed in such a way as to emphasize the correspondence with the usual presentation in textbooks (although in this case the conclusion is above the horizontal line).

Notice how rules `VarR` and `OpR` have suffered a modification in order to avoid patterns like `rho(x)` or `Ap(op, v, v')` in the conclusion (lefthand side of the rewrite rule). They are instead used in a side condition.

In this way, we start with a transition to be proved valid and we work backwards using the rewriting process, maintaining the set of transitions that have to be fulfilled in order to prove the correctness of the initial transition. The initial transition can be rewritten to the empty set if and only if it is a valid transition in the operational semantics.

However, we found two problems while working with this approach. The first one is that sometimes new variables appear in the premises which are not present in the conclusion (for example, in rule `OpR` above `v` and `v'` are new variables in the righthand side). Rules of this kind cannot be directly used by the Maude default interpreter; they can only be used at the metalevel using a strategy to instantiate the extra variables. The second problem is that sometimes several rules can be applied to rewrite a judgement, but in general, not all of the possibilities lead to an empty set of judgements. So we have to deal with the whole computation tree of possible rewrites of a judgement, searching to see if one of the branches leads to `emptyJS`.

In [71] we presented general solutions to these problems by modifying the semantics representation (at the object level) and controlling the rewriting process by means of a strategy at the metalevel.

The presence of new variables was solved by using the concept of *explicit metavariables* presented in [61] in a very similar context, which make explicit the lack of knowledge that new variables in the righthand side of a rewrite rule represent. The semantics with explicit metavariables has to bind them to concrete values when these values become known. Thus, we introduced in the semantics representation mechanisms to deal with these bindings and propagate them to other judgements where the bound metavariable may be present. The modified representation also has rules with new variables in the righthand side, but now they are localized. The strategy that controls the rewriting process (see below) is in charge of instantiating these variables in order to build *new* metavariables.

The problem of non-deterministic application of rewrite rules was solved by a general search strategy defined at the metalevel. The strategy traverses the conceptual tree of all possible rewrites of a term, built by using the rewrite rules representing the semantics, searching for the term representing the empty set of judgements. If it is found, the transition represented at the root of this tree is a valid semantic transition.

In [71], for the CCS case, we also extended the operational semantics implementation by including metavariables as processes (before that, we only needed metavariables as actions). If we start the search strategy with a judgement where the process in the righthand side of the CCS transition is a metavariable, like in `P -- a --> ?P`, and the search reaches the empty set, then the metavariable `?P` has to be bound to one of the one-step successors of the process in the lefthand side, `P`, after performing action `a`. By extending the search strategy to find not only the first way to reach the empty set, but all the possible ways, we implemented an operation that returns all the successors of a process after performing a given action. This operation was then used to implement the Hennessy-Milner modal logic for CCS processes [63], by following the same techniques for dealing with new variables and with non-determinism as in the CCS semantics, that is, by defining rewrite rules that rewrite a modal logic judgement $P \models \Phi$ into the set of judgements which have to be satisfied (as specified by the logic's semantics) [71]. The search strategy has to be used again, now to check if a modal logic judgement is true. Each time the strategy is used, the module with the rewrite rules that defines the search tree has to be metarepresented. Thus, we obtained three levels of representation. The CCS semantics rules are in the first level. They are controlled by the search strategy at the second level, where the operation that returns all the successors of a process and the modal logic semantics are defined. Finally, the modal logic semantics is controlled by the search strategy at the third level.

8.1 Comparison with the transitions as rewrites approach

The transitions as judgements approach has a marked role as *prover*, in which the rewriting process corresponds to finding a proof of the initial judgement being rewritten. Intuitively, the idea is that we start with a transition to be proved valid and we work backwards using the rewriting process in a goal-directed way, maintaining the set of transitions that have to be fulfilled in order to prove the correctness of the initial transition. When this set is empty we can conclude that the initial transition is a correct transition, that is, the initial transition can be rewritten to the empty set if and only if it is a valid transition in the operational semantics.

On the other hand, the transitions as rewrites approach leads us to implementations with a marked role as *interpreters*, where given an environment and an expression, rules rewrite them to the value they are reduced by the operational semantics.

However, Maude allows us to use either implementation in either role, like a prover or like an interpreter. The use of metavariables to solve the problem of new variables in the righthand side of a rule (as mentioned above), as well as the adaptation of the representation to deal with those metavariables, has as a lateral effect that we can use the obtained representation using the transitions as judgements approach like an interpreter that calculates, given an expression, the value to which it is evaluated, without having to start with a complete judgement that includes this value. If, for example, we want to evaluate expression `e` with a set of declarations `D` and in an environment `rho`, we can rewrite the judgement `D,rho |- e ==>A ?('result)` where `?('result)` is a metavariable. In the rewriting process of this judgement, the metavariable will be bound to the result of evaluating `e`. In Section 3.2 we saw how the `search` command is useful to use the transitions as rewrites implementation as a prover, where we can check if a given judgement is derivable from the semantics rules.

In our opinion the implementation following the transitions as rewrites approach has several advantages. This implementation is closer to the mathematical, logical presentation of the semantics. An operational semantics rule establishes that the transition in the conclusion is possible if the transitions in the premises are possible, and that is precisely the interpretation of a conditional rewrite rule with rewrite conditions. The alternative approach needs auxiliary structures like the multisets of judgements to be proved valid and mechanisms like the generation of new metavariables and their propagation when their concrete values become known. This forced us to implement at the metalevel a search strategy that checks if a given multiset can be reduced to the empty set and generates new metavariables each time they are needed. It is the necessity of new metavariables what makes the strategy unavoidable. We could not use the `search` command of Maude 2.0, because it cannot handle rewrite rules with new variables in the righthand side whenever they are not bound in any of the conditions, and that is what happens in this kind of implementations [71]. With the transitions as rewrites approach the necessity of searching appears in the rewrite conditions, but the Maude 2.0 system solves the problem, because it is able to handle these conditions together with new variables bound in some condition.

There are also differences found in the things that are done at the object level (level of the semantics representation) and at the metalevel (by using reflection). In the transitions as judgements approach, the search strategy traverses the conceptual tree with all the possible rewrites of a term, moving continuously between the object level and the metalevel. In the implementations described in this paper, the search occurs completely at the object level, which makes it considerably faster and simpler.

9 Related work

We can find in the literature several works dedicated to the representation and implementation of operational semantics. We cite here some of the most related to our work.

Probably the work most closely related to ours is the one by Christiano Braga in his PhD. thesis [4], where he describes an interpreter for MSOS specifications [5] in the context of Peter Mosses's modular structural operational semantics [53]. In the interpreter implementation the approach of transitions as rewrites is used, by making an extension of Maude implemented using the reflective features of Maude itself that allows conditional rules with rewrites in the conditions. We have used Maude 2.0, obtaining a considerable efficiency enhancement. This line of work has been continued very recently by integrating some of the methods that we have proposed in this paper and others in order to develop a general method to achieve modularity of semantic definitions of programming languages specified as rewrites theories [6].

As we have already mentioned at the beginning of this paper, there is a rich tradition of using rewriting logic to give semantic definitions for languages using a variety of styles, including the lambda calculus [48, 60], Prolog and languages based on narrowing like BABEL [74], the UNITY language [50], the π-calculus [73, 60, 65], the concurrent logic programming language GAEA [43], the programming language for active networks PLAN [75, 62], a UML metamodel [66, 30, 31], the specification language for cryptographic protocols CAPSL [21], the mobile agents system DaAgent [1], the Maude extension for mobile computations Mobile Maude [25], and the Resource Description Framework (RDF) for the semantic web [3]. For a more exhaustive bibliography about this subject we refer to the paper [49]. With the exception of the more recent paper [65], that applies to the π-calculus the techniques that we have applied to CCS in Section 6, none of the other papers use rewrite rules with rewrites in the conditions, because those rules could not be executed in previous versions of Maude.

Perhaps one of the first attempts to get direct implementations for operational semantics was Typol [23], a formal language for representing inference rules and operational semantics. Typol programs were compiled to Prolog to build executable type ckeckers and interpreters from their specifications [22]. Although some of our implementations follow in some respects the logic programming style, a great advantage of using Maude consists in the possibility of working, on the one hand, with data types defined by the user, and on the other hand, with algebraic specifications modulo equational axioms. Moreover, we could use other strategies different from depth-first search, even keeping the same underlying specification.

Some disadvantages of Typol are its inefficiency and the fact that the implementation of specifications of structural operational semantics in Prolog is not attractive, due to the lack of an appropriate type system in Prolog (some authors have used the higher order language λProlog [28] to avoid this problem). For all these reasons, the language RML (Relational Meta-Language) [56, 57] was designed, a language for the executable specification of natural semantics. In this study, properties of natural semantics specifications were identified as determinable in a static way, allowing some optimizations in the implementation. RML has a strong type system in the style of Standard ML, and it supports inference rules like those in natural semantics, and data type definitions by means of structural induction. Specifications in RML are translated into an intermediate representation, which can then be easily optimized and implemented, following the style CPS (*Continuation-Passing Style*). This intermediate representation is finally compiled to efficient C code.

Theorem provers like Isabelle/HOL [55] or Coq [42] have also been used to build models of languages from their operational semantics. Isabelle/HOL has been used by Nipkow [54] to formalize operational and denotational semantics of programming languages. Other logical frameworks and theorem provers have also been used to represent inference systems. The interactive proof development environment Coq [42], based on the Calculus of Constructions extended with inductive types, has been used to represent the π-calculus [32, 40] and the μ-calculus [59] applied to CCS. Coq is used to encode natural semantics in [64]. In these works the approach is different from ours, since instead of obtaining executable representations, they focus on getting models with which metaproperties can be verified.

10 Conclusions and future work

In this paper we have shown how the *transitions as rewrites* approach can be used to implement a wide variety of structural operational semantics in Maude, an *executable semantic framework*.

Sometimes we have needed to make precise some details in the mathematical definition of a semantics, as when an ellipsis "..." appears in the premises of a semantic rule. The Maude facilities for defining syntactic operators, including the associativity and identity attributes, and pattern matching modulo these properties, have allowed us to make precise those details in a clear and easy way, resolving, for example, the non-deterministic choice of one of the arguments of a function call to be reduced. We have also been able to define, at the same level of the semantics, the syntactic substitution operation, used in several of the semantics definitions, including the generation of new variables (not ocurring in the expression being evaluated) in order to avoid free variable capture.

In the Mini-ML example, we have found a semantic rule (that one corresponding to the letrec operator) that cannot be implemented directly. The problem is that the original semantics is not so *operational* as it might seem, since in some moment we need to guess the value we want to calculate in order to infer it. Certainly, this kind of semantic rule is not usual, because its meaning is not intuitive. The solution consists in finding an alternative semantic rule, an implementable one.

After successfully representing a semantics, we have shown how the Maude commands can be used to obtain several kinds of information. When a higher control of the possible transitions is needed, as in the implementation of the Hennessy-Milner modal logic for CCS, reflection and the `META-LEVEL` predefined module provide a valuable tool. Moreover, we can implement complete tools that execute the user language and hide the concrete representation of the semantics, as we have done for Full LOTOS.

We want to study now how to analyze and prove properties about the obtained semantics representations, such as confluence or termination. These properties do not refer to concrete programs written in the language whose semantics is represented; they are instead metaproperties applicable to every program in general. Most of them are proved by structural induction on the rules that define the semantics [36]. In this respect, we intend to study extensions of the ITP theorem prover [13].

Based on the symbolic semantics for LOTOS used in Section 7, a symbolic bisimulation [10] and a modal logic FULL [9] have been defined. We plan to extend our tool so that we can check if two processes are bisimilar, or if a process satisfies a given modal logic formula. We have already implemented a subset of FULL without data values (following the same techniques we use to implement the Hennessy-Milner modal logic for CCS processes in Section 6.4), and we have integrated it with our tool. The part of the logic with data values deserves more study, and we think that some kind of theorem proving will be needed. Rewriting logic and Maude have been proved highly valuable also for these subjects [12].

In joint work with José Meseguer, we are designing a strategy language that allows specifying which strategy has to be used in order to rewrite a term. Basic strategy expresions will be rule labels, meaning that a particular rule has to be applied. When the rule is conditional, with rewrites in the conditions, the strategy expression may describe how each of the conditions has to be resolved. If we have a rewrite condition `t => t'` the strategy would say how `t` has to be rewritten in order to find a term that matches `t'`; in particular, it will say which rewrite rules can be applied in the rewriting process. Basic strategies are combined to build greater strategies by union, concatenation, disjunction (by means of a generalized if-then-else), iteration, etc.

With this language we will be able to solve (in a different, perhaps more elegant way) the problems we found in Section 3.3 (when defining the reflexive, transitive closure of the computation semantics relation for *Fpl*) and in Section 6.2 (when defining the reflexive, transitive closure of the CCS transition relation). We foresee that this strategy language will be quite useful in the implementation and execution of the parallel functional language Eden [7, 39], where we intend to study how different execution strategies influence the semantics properties.

This strategy language will be applicable not only to operational semantics representations, but to executable specifications in general.

Acknowledgments

We would like to thank Christiano Braga, Roberto Bruni, Francisco Durán, Steven Eker, David de Frutos, José Meseguer, Fernando Orejas, Yolanda Ortega, Miguel Palomino, and Carron Shankland for all their very helpful comments on previous presentations of this work, and José Meseguer for encouraging and supporting all our research on executable semantic frameworks.

References

[1] J. V. Baalen, J. L. Caldwell, and S. Mishra. Specifying and checking fault-tolerant agent-based protocols using Maude. In J. Rash, C. Rouff, W. Truszkowski, D. Gordon, and M. Hinchey, editors, *First International Workshop, FAABS 2000, Greenbelt, MD, USA, April 2000. Revised Papers*, volume 1871 of *Lecture Notes in Artificial Intelligence*, pages 180–193. Springer, 2000.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[3] M. Bradley, L. Llana, N. Martí-Oliet, T. Robles, J. Salvachua, and A. Verdejo. Transforming information in RDF to rewriting logic. In R. Peña, A. Herranz, and J. J. Moreno, editors, *Segundas Jornadas sobre Programación y Lenguajes. (PROLE 2002)*, pages 167–182, 2002.

[4] C. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics.* PhD thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brazil, Sept. 2001.

[5] C. Braga, E. Hermann Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In T. Rus, editor, *AMAST: 8th International Conference on Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2000.

[6] C. Braga and J. Meseguer. Modular rewriting semantics of programming languages. Manuscript, University of Illinois at Urbana-Champaign, 2003.

[7] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Eden – The paradise of functional concurrent programming. In *Euro-Par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 710–713. Springer, 1996.

[8] J. Bryans and C. Shankland. Implementing a modal logic over data and processes using XTL. In Kim et al. [46], pages 201–218.

[9] M. Calder, S. Maharaj, and C. Shankland. An adequate logic for Full LOTOS. In J. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 384–395. Springer, 2001.

[10] M. Calder and C. Shankland. A symbolic semantics and bisimulation for Full LOTOS. In Kim et al. [46], pages 184–200.

[11] G. Carabetta, P. Degano, and F. Gadducci. CCS semantics via proved transition systems and rewriting logic. In Kirchner and Kirchner [47], pages 253–272.

[12] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Publications, 2000.

[13] M. Clavel. The ITP tool. In A. Nepomuceno, J. F. Quesada, and J. Salguero, editors, *Logic, Language and Information. Proceedings of the First Workshop on Logic and Language*, pages 55–62. Kronos, 2001.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude as a metalanguage. In Kirchner and Kirchner [47].

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual. Version 1.0.* Computer Science Laboratory, SRI International, June 2003. `http://maude.cs.uiuc.edu/manual`.

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications 14th International Conference, RTA 2003 Valencia, Spain, June 2003 Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2003.

[18] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic. In Gadducci and Montanari [33], pages 63–78.

[19] R. Cleaveland and S. T. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, 2002.

[20] P. Degano, F. Gadducci, and C. Priami. A causal semantics for CCS via rewriting logic. *Theoretical Computer Science*, 275(1–2):259–282, 2002.

[21] G. Denker and J. Millen. CAPSL integrated protocol environment. In D. Maughan, G. Koob, and S. Saydjari, editors, *Proceedings DARPA Information Survivability Conference and Exposition, DISCEX 2000, Hilton Head Island, South Carolina, January 25-27, 2000*, pages 207–222. IEEE Computer Society Press, 2000. `http://schafercorp-ballston.com/discex/`.

[22] T. Despeyroux. Executable specification of static semantics. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 215–233. Springer, 1984.

[23] T. Despeyroux. TYPOL: A formalism to implement natural semantics. Research Report 94, INRIA, 1988.

[24] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[25] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In D. Kotz and F. Mattern, editors, *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zurich, Switzerland, September 13–15, 2000, Proceedings*, volume 1882 of *Lecture Notes in Computer Science*. Springer, Sept. 2000.

[26] H. Eertink. Executing LOTOS specifications: the SMILE tool. In T. Bolognesi, J. Lagemaat, and C. Vissers, editors, *LotoSphere: Software Development with LOTOS*. Kluwer Academic Publishers, 1995.

[27] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[28] A. Felty, E. Gunter, J. Hannan, D. Miller, G. Nadathur, and A. Scedrov. Lambda Prolog: An extended logic programming language. In E. Lusk and R. Overbeek, editors, *Proceedings on the 9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 754–755. Springer, 1988.

[29] J. C. Fernández, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: a protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer, 1996.

[30] J. L. Fernández and A. Toval. Can intuition become rigorous? Foundations for UML model verification tools. In F. M. Titsworth, editor, *International Symposium on Software Reliability Engineering*, pages 344–355, San José, California, Oct. 2000. IEEE Press.

[31] J. L. Fernández and A. Toval. Seamless formalizing the UML semantics through metamodels. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, pages 224–248. Idea Group Publishing, 2001.

[32] M. M. Furio Honsell and I. Scagnetto. π-calculus in (co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

[33] F. Gadducci and U. Montanari, editors. *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. `http://www.elsevier.nl/locate/entcs/volume71.html`.

[34] B. Ghribi and L. Logrippo. A validation environment for LOTOS. In A. Danthine and G. Leduc, editors, *Protocol Specification, Testing, and Verification XIII*, pages 93–108. Norht-Holland, 1993.

[35] R. Guillemot, M. Haj-Hussein, and L. Logrippo. Executing large LOTOS specifications. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 399–410. North-Hollland, 1988.

[36] M. Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Willey & Sons, 1990.

[37] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[38] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, Jan. 1985.

[39] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters*, 12(2):211–228, 2002.

[40] D. Hirschkoff. A full formalisation of π-calculus theory in the calculus of constructions. In *Proc. 10th International Theorem Proving in Higher Order Logic Conference*, volume 1275 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 1997.

[41] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[42] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant: a tutorial: version 7.2. Technical Report 256, INRIA, 2002.

[43] H. Ishikawa, J. Meseguer, T. Watanabe, K. Futatsugi, and H. Nakashima. On the semantics of GAEA — An object-oriented specification of a concurrent reflective language in rewriting logic. In *Proceedings IMSA'97*, pages 70–109. Information-Technology Promotion Agency, Japan, 1997.

[44] ISO/IEC. *LOTOS—A formal description technique based on the temporal ordering of observational behaviour.* International Standard 8807, International Organization for standardization — Information Processing Systems — Open Systems Interconnection, Geneva, Sept. 1989.

[45] G. Kahn. Natural semantics. Technical Report 601, INRIA Sophia Antipolis, Feb. 1987.

[46] M. Kim, B. Chin, S. Kang, and D. Lee, editors. *Proceedings of FORTE 2001, 21st International Conference on Formal Techniques for Networked and Distributed Systems.* Kluwer Academic Publishers, 2001.

[47] C. Kirchner and H. 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.* Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[48] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002.

[49] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

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

[51] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[52] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[53] P. Mosses. Foundations of modular SOS. In M. Kutylowski, L. Pacholksi, and T. Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99 Szklarska Poreba, Poland, September 6–10, 1999, Proceedings*, volume 1672 of *Lecture Notes in Computer Science*, pages 70–80. Springer, 1999. The full version appears as Technical Report RS-99-54, BRICS, Dept. of Computer Science, University of Aarhus.

[54] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10(2):171–186, 1998.

[55] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

[56] M. Pettersson. RML — A new language and implementation for natural semantics. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94*, volume 844 of *Lecture Notes in Computer Science*, pages 117–131. Springer, 1994.

[57] M. Pettersson. A compiler for natural semantics. In T. Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 1996.

[58] J. C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[59] C. Sprenger. A verified model checker for the modal μ-calculus in Coq. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1998.

[60] M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71–92. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[61] M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic. In *Proc. of LFM'99: Workshop on Logical Frameworks and Meta-Languages*, Paris, France, Sept. 1999.

[62] M.-O. Stehr and C. L. Talcott. PLAN in Maude: Specifying an active network programming language. In Gadducci and Montanari [33], pages 195–215.

[63] C. Stirling. Modal and temporal logics for processes. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure vs Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 149–237. Springer, 1996.

[64] D. Terrasse. Encoding natural semantics in Coq. In V. S. Alagar, editor, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 1995.

[65] P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. In Gadducci and Montanari [33], pages 217–237.

[66] A. Toval and J. L. Fernández. Formally modeling UML and its evolution: A holistic approach. In S. F. Smith and C. L. Talcott, editors, *Proceedings IFIP Conference on Formal Methods for Open Object-Based Distributed Systems IV, FMOODS 2000, September 6–8, 2000, Stanford, California, USA*, pages 183–206. Kluwer Academic Publishers, 2000.

[67] K. Turner. *Using Formal Description Techniques – An Introduction to Estelle, LOTOS and SDL.* John Wiley and Sons Ltd., 1992.

[68] A. Verdejo. Building tools for LOTOS symbolic semantics in Maude. In D. Peled and M. Vardi, editors, *Formal Techniques for Networked and Distributed Systems — FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 2002 Proceedings*, volume 2529 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2002.

[69] A. Verdejo. A tool for Full LOTOS in Maude. Technical Report 123-02, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Apr. 2002. `http://www.ucm.es/sip/alberto`.

[70] A. Verdejo. *Maude como marco semántico ejecutable.* PhD thesis, Universidad Complutense de Madrid, 2003.

[71] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In T. Bolognesi and D. Latella, editors, *Formal Methods For Distributed System Development. FORTE/PSTV 2000 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX) October 10–13, 2000, Pisa, Italy*, pages 351–366. Kluwer Academic Publishers, 2000.

[72] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In Gadducci and Montanari [33], pages 239–257.

[73] P. Viry. Input/output for ELAN. In J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 51–64. Elsevier, Sept. 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[74] M. Vittek. *ELAN: Un Cadre Logique pour le Prototypage de Langages de Programmation avec Contraintes.* PhD thesis, Université Henri Poincaré – Nancy I, Nov. 1994.

[75] B.-Y. Wang, J. Meseguer, and C. A. Gunter. Specification and formal analysis of a PLAN algorithm in Maude. In P.-A. Hsiung, editor, *International Workshop on Distributed System Validation and Verification*, pages 49–56, 2000.