A Maude Tutorial*

M. Clavel
Univ. de Navarra, Spain

F. Durán
Univ. de Málaga, Spain

S. Eker
SRI International, CA, USA

P. Lincoln
SRI International, CA, USA

N. Martí-Oliet
Univ. Complutense Madrid, Spain

J. Meseguer
SRI International, CA, USA

J. F. Quesada
CICA, Sevilla, Spain

March 2000

Contents

*Corrections and comments on this document should be addressed to `narciso@sip.ucm.es`.

1 Introduction

Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Rewriting logic [21] is a logic of concurrent change that can naturally deal with state and with highly nondeterministic concurrent computations. It has good properties as a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency [23]. In particular, it supports very well concurrent *object-oriented* computation. This is reflected in Maude's design by providing special syntax for *object-oriented modules.* Since the computational and logical interpretations of rewriting logic are like two sides of the same coin, the same reasons making it a good semantic framework at the computational level make it also a good *logical framework* at the logical level, that is, a *metalogic* in which many other logics can be naturally represented and implemented [20]. Consequently, some of the most interesting applications of Maude are *metalanguage* applications, in which Maude is used to create executable environments for different logics, theorem provers, languages, and models of computation. In this regard, exploiting the fact that rewriting logic is reflective [10, 4], a key distinguishing feature of Maude is its systematic and efficient use of *reflection*, a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications.

This tutorial consists in a sequence of examples, each one introducing several features of the Maude language in an incremental way. We begin with many-sorted especifications with conditional equations and successively introduce modularization, subsorts, equational attributes, memberships, and parameterization. Then we move from the static world of equational logic to the dynamic world of rewriting logic and in particular of object-oriented systems. The most complex level we reach is the introduction of reflection, metaprogramming, and internal strategies.

The Maude system, its documentation [6], a collection of examples, some case studies, and related papers are available on the Maude web page at `http://maude.csl.sri.com`.

2 Many-sorted equational specifications

Algebraic specifications are used to declare different kinds of data together with the operations that act upon them. The behavior of operations is described by means of conditional equations.

It is useful to distinguish two kinds of operations: constructors, that are used to construct or generate the data, and the remaining operations (that can also be classified as modifiers or as observers according to whether the result type is the same as the type of the data being defined or another). However, it must be emphasized that this classification belongs to the methodology but not to the logic on which the specifications are based. Thus, from the logical point of view, all operations enjoy the same status.

The version of equational logic that provides the logical basis for our algebraic specifications is *membership equational logic*, but instead of presenting it directly we will introduce all its different features incrementally. For this reason, we start with its well-known many-sorted equational sublogic.

2.1 Signatures, terms, and equations

The first thing a specification needs to declare are the types (that in the algebraic specification community are usually called *sorts*) of the data being defined and the corresponding operations. A many-sorted *signature* (S, Σ) consists of a sort set S and an $S^* \times S$-sorted family $\Sigma = \{\Sigma_{\overline{s},s} \mid \overline{s} \in S^*, s \in S\}$ of sets of operation symbols. When $\sigma \in \Sigma_{\overline{s},s}$, we say that σ has *rank* $\langle \overline{s}, s \rangle$, *arity* $\overline{s}$, and *value sort* (or *coarity*) s. In particular, a constant of sort s is identified with an operation with rank $\langle \varepsilon, s \rangle$ where the arity ε denotes the empty string in S^*; in this way, a constant is viewed as an operation without arguments.

With the declared operations we can construct *terms* to denote the data being specified. Since all the operations are strongly typed, terms are also typed, and can be classified according to the sort of the denoted data. Moreover, terms can have *variables*, either to represent unspecified fragments or to range over several possibilities.

Given a many-sorted signature (S, Σ) and an S-sorted family $X = \{X_s \mid s \in S\}$ of pairwise disjoint sets of variables, also disjoint from Σ, the S-sorted set of terms $\mathcal{T}_\Sigma(X) = \{\mathcal{T}_{\Sigma,s}(X) \mid s \in S\}$ is inductively defined by the following conditions:

1. $X_s \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$; that is, variables are terms,
2. $\Sigma_{\varepsilon,s} \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$; that is, constants are terms,
3. If $\sigma \in \Sigma_{\overline{s},s}$ and $t_i \in \mathcal{T}_{\Sigma,s_i}(X)$ $(i = 1, \ldots, n)$, where $\overline{s} = s_1 \ldots s_n \neq \varepsilon$, then $\sigma(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,s}(X)$; that is, an operation symbol(syntactically) applied to terms of the appropriate sorts produces a new term.

A finite S-sorted set of variables X is represented as a list $x_1 : s_1, \ldots, x_n : s_n$, or more concisely $\overline{x} : \overline{s}$ (where all variables x_i are distinct).

A Σ-*equation* is an expression $(\overline{x} : \overline{s})\, l = r$, where $\overline{x} : \overline{s}$ is a (finite) set of variables, and l and r are terms in $\mathcal{T}_{\Sigma,s}(\overline{x} : \overline{s})$ for some sort s.

A *conditional* Σ-*equation* is an expression

$$(\overline{x} : \overline{s})\, l = r \;\; \textit{if} \;\; u_1 = v_1, \ldots, u_n = v_n$$

where $(\overline{x} : \overline{s})\, l = r$, and $(\overline{x} : \overline{s})\, u_i = v_i$, for $i = 1, \ldots, n$, are Σ-equations .

A many-sorted specification consists of a signature (S, Σ) and a set E of (conditional) Σ-equations.

The semantics of such a specification is defined by algebras. A many-sorted (S, Σ)-algebra $\mathbf{A}$ consists of a carrier set A_s for each $s \in S$ and a function $A_\sigma^{\overline{s},s} : A_{\overline{s}} \to A_s$ for each operation symbol $\sigma \in \Sigma_{\overline{s},s}$. Algebras are related by means of homomorphisms, that is, maps that preserve the algebra structure. It is possible to define inductively the meaning of a term in an algebra and then satisfaction of a (conditional) equation by an algebra. The *loose semantics* of a many-sorted specification (S, Σ, E) is the set of (S, Σ)-algebras that satisfy all the (conditional) equations in E, but we are usually interested in the so-called *initial semantics* given by a particular algebra in this class (up to isomorphism). A possible concrete representation $\mathcal{T}_{\Sigma,E}$ of such an initial algebra is obtained by imposing a congruence relation on the *term algebra* $\mathcal{T}_\Sigma$ whose carrier sets are the sets of *ground* terms, that is, terms without variables. Two terms are identified by this congruence if an only if they have the same meaning in all algebras in the loose semantics, but it is also possible to define this congruence proof-theoretically by means of some deduction system $\vdash$ for equational logic, so that the *congruence class* of a term t is given by

$$[t]_E = \{t' \in \mathcal{T}_\Sigma \mid E \vdash_\Sigma t = t'\}.$$

Mathematically, there is a unique *quotient homomorphism* $q_E : \mathcal{T}_\Sigma \longrightarrow \mathcal{T}_{\Sigma,E}$, sending a term t to its corresponding congruence class $[t]_E$.

There are several books on equational specification of abstract data types, where the reader can find much more information on many-sorted equational specifications and their semantics [14, 19, 26, 1].

2.2 Matching, rewriting, and equational simplification

The mathematical semantics of an equational specification based on many-sorted algebras (either with loose or initial semantics) provides justification for the statement that a specification indeed specifies some model we have in mind. Although this semantics can be used to answer concrete questions about, for example, whether two terms have the same meaning, it is more convenient to rely on more syntactic

means, that also provide more opportunities for efficient mechanization. As we have already mentioned, there is a proof theory defining a deduction relation for (conditional) equations. Under certain conditions that we summarize in this section, equational deduction can be mechanized by means of the notions of matching and rewriting, where equations are oriented as reduction rules from left to right.

Given S-sorted families of variables X and Y for a signature (S, Σ), a substitution is a sort-preserving map $\sigma : X \to \mathcal{T}_\Sigma(Y)$; such a map extends uniquely to a homomorphism over terms $\mathcal{T}_\Sigma(X) \to \mathcal{T}_\Sigma(Y)$ also denoted σ.

Given a term $t \in \mathcal{T}_\Sigma(X)$, corresponding to the lefthand side of an oriented equation, and a subject term $u \in \mathcal{T}_\Sigma(Y)$, we say that t *matches* u if there is a substitution σ such that $\sigma(t) \equiv u$, that is, $\sigma(t)$ and u are syntactically equal terms.

The first condition we require on an (oriented) Σ-equation $(\overline{x} : \overline{s})\, l = r$ is that all variables in the righthand side r also appear among the variables of the lefthand side l; furthermore, in the case of conditional equations all variables occurring in the conditions must also appear among the variables of l. Under this assumption, a term t *rewrites* to a term t' using such an equation if there is a subterm $t|_p$ of t such that l matches $t|_p$ via a substitution σ and t' is obtained from t by replacing the subterm $t|_p \equiv \sigma(l)$ with the term $\sigma(r)$. This is denoted $t \to_E t'$ when the possible equations for rewriting are chosen in the set E. The reflexive and transitive closure of the relation $\to_E$ is denoted $\to_E^*$.

A set of equations E is *confluent* (or *Church-Rosser*) when the result of rewriting a term is unique in the following sense: if $t \to_E^* t_1$ and $t \to_E^* t_2$, then there exists a term t' such that $t_1 \to_E^* t'$ and $t_2 \to_E^* t'$.

A set of equations E is *terminating* when there is no infinite sequence of rewriting steps $t_0 \to_E t_1 \to_E t_2 \to_E \ldots$

If E is both confluent and terminating, a term t can be reduced to a unique *normal form* $t{\downarrow_E}$, that is, to a term that can no longer be rewritten. Therefore, in order to check semantic equality of two terms $t = t'$ (equivalently, that they belong to the same congruence class), it is enough to check that their respective normal forms are equal, $t{\downarrow_E} = t'{\downarrow_E}$, but, since normal forms cannot be rewritten anymore, the last equality is just syntactic coincidence: $t{\downarrow_E} \equiv t'{\downarrow_E}$.

In the case of conditional equations, and assuming confluence and termination, an equation

$$(\overline{x} : \overline{s})\, l = r \;\; \textit{if} \;\; u_1 = v_1, \ldots, u_n = v_n$$

can be used to rewrite a term t to a term t' if there is a subterm $t|_p$ of t such that l matches $t|_p$ via a substitution σ, and $\sigma(u_i){\downarrow_E} \equiv \sigma(v_i){\downarrow_E}$ for $i = 1, \ldots, n$; then, as in the unconditional case, t' is obtained from t by replacing the subterm $t|_p \equiv \sigma(l)$ with the term $\sigma(r)$.

Equational specifications in Maude, introduced by the keyword `fmod` (*functional module*), are assumed to be confluent and terminating, and their operational semantics is *equational simplification*, that is, rewriting of terms until a normal form is obtained. Notice that the system does not check the confluence and termination properties, so that they are left to the user's responsibility.

For more details on matching and rewriting, we refer the reader to the recent book [2].

2.3 Boolean values

We begin with a simple classical example, defining the Boolean values and some logical operations on them (negation, conjunction and disjunction). To construct the data it is enough to have two different constants, `true` and `false`. The other operations are equationally defined by structural induction over the constructors, but in the case of conjunction and disjunction it is enough to have induction over the first argument.

Note that we use mixfix syntax, which is more convenient from the user's point of view than being forced to use always prefix syntax. The mixfix operation declaration uses the underbar character to denote the places of arguments, and thus it must have the same number of underbars as sorts appearing in the arity of the operation.

Moreover, instead of declaring variables in each equation, there is a global variable declaration that affects all equations in the module.

```
fmod BOOLEAN is
  sort Bool .
  op true : -> Bool [ctor] .
  op false : -> Bool [ctor] .
  op not_ : Bool -> Bool .
  op _and_ : Bool Bool -> Bool .
  op _or_ : Bool Bool -> Bool .
  var A : Bool .
  eq not true = false .
  eq not false = true .
  eq true and A = A .
  eq false and A = false .
  eq true or A = true .
  eq false or A = A .
endfm
```

Notice the use of the attribute `ctor` to indicate the operations used as constructors, the two constants `true` and `false` in this example.

Although the equations above define completely the behavior of all the operations, it is possible to add equations expressing additional properties (satisfied by the initial algebra), like associativity or commutativity, for example:

```
  vars A B C : Bool .
```



```
  eq A and B = B and A .
  eq (A and B) and C = A and (B and C) .
```

Adding these equations does not change the initial semantics of the specification, because the initial algebra of `BOOLEAN` (isomorphic to the usual two-value Boolean algebra) indeed satisfies them. However, it changes the loose semantics, by discarding all those algebras that do not satisfy them. More important, however, is the possible bad behavior from the operational semantics point of view of equations such as commutativity, because it causes nontermination of equational simplification. In Section 5.1, we will see a possible solution that allows the use of this kind of nonterminating equations.

2.4 Natural numbers

Our second example is another classic, the natural numbers in Peano notation. In this case, it is not practical to have an infinite set of different constants to denote the values. Instead, we can use a constant `0` and a successor operation `s` (since there is no underbar in its declaration, this operation has the usual prefix syntax with parentheses around the argument for application).

Since successor is injective and zero is not the successor of any number, these constructors (notice the corresponding attribute) are *free*, that is, all the terms generated by them denote different data, and thus there is no identification among them.

```
fmod BASIC-NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
endfm
```

In this example the only other operation defined over natural numbers is addition, written in mixfix syntax, and equationally defined by structural induction over the two constructors in the first argument.

Again, it would be possible to add equations expressing additional properties of addition, like associativity and/or commutativity. This does not change the initial semantics (isomorphic to the usual set of natural numbers with addition), but it does change the loose semantics, since there are "nonstandard" models of `BASIC-NAT` for which addition is neither associative nor commutative. However, it must be emphasized that the thing to watch out for is the possible bad behavior from the operational semantics point of view of equations such as commutativity, as we have already pointed out before.

2.5 Operations on natural numbers

We want to define the natural numbers with more arithmetic operations on them, and in particular with some comparison operations. Although one might consider signatures with functions and predicates, we take the functional view in which predicates are specified as operations having `Bool` as value sort. This is our first example of operations relating more than just one data type; therefore, the specification must define all the necessary data types and all their corresponding operations.

The new arithmetic operations with respect to the ones in previous examples are multiplication and subtraction over natural numbers. The former is equationally defined by induction on the first argument, while the latter needs induction on the first argument and then induction on the second. Exactly the same case analysis is necessary to define the comparison predicate `_<=_`. Finally, we use Boolean negation to define `_>_` in terms of `_<=_`.

```
fmod NAT+OPS is
  *** copy of BOOLEAN
  sort Bool .
  op true : -> Bool [ctor] .
  op false : -> Bool [ctor] .
  op not_ : Bool -> Bool .
  op _and_ : Bool Bool -> Bool .
  op _or_ : Bool Bool -> Bool .
  var A : Bool .
  eq not true = false .
  eq not false = true .
  eq true and A = A .
  eq false and A = false .
  eq true or A = true .
  eq false or A = A .

  *** copy of BASIC-NAT
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
```



```
  op _*_ : Nat Nat -> Nat .
  op _-_ : Nat Nat -> Nat .
  op _<=_  : Nat Nat -> Bool .
  op _>_ : Nat Nat -> Bool .

  eq 0 * N = 0 .
  eq s(M) * N = (M * N) + N .
  eq 0 - N = 0 .
  eq s(M) - 0 = s(M) .
  eq s(M) - s(N) = M - N .
  eq 0 <= N = true .
  eq s(M) <= 0 = false .
  eq s(M) <= s(N) = M <= N .
  eq M > N = not (M <= N) .
endfm
```

3 Modularization

Each of the specifications introduced in the previous section is given as a simple module, that is, they are basic unstructured specifications. Here we introduce the first structuring mechanisms, providing two operations to import one specification into another one.

3.1 Importation

The module `NAT+OPS` in Section 2.5 uses the sorts and operations in the modules of previous examples `BOOLEAN` and `BASIC-NAT`. In the above version of `NAT+OPS` the solution adopted is to repeat all the declarations of sorts and operations, as well as all the corresponding equations. Obviously, even with the availability of the "cut and paste" facility in text editors, it is very useful to have a structural view of inclusion dependencies among modules. An equivalent presentation of the module `NAT+OPS` uses Maude's importation mechanism `protecting` (abbreviated to `pr`), only having to declare the new operations and their equations.

```
fmod NAT+OPS is
  protecting BOOLEAN .
  protecting BASIC-NAT .

  op _*_ : Nat Nat -> Nat .
  op _-_ : Nat Nat -> Nat .
  op _<=_ : Nat Nat -> Bool .
```

```
  op _>_ : Nat Nat -> Bool .

  vars N M : Nat .
  eq 0 * N = 0 .
  eq s(M) * N = (M * N) + N .
  eq 0 - N = 0 .
  eq s(M) - 0 = s(M) .
  eq s(M) - s(N) = M - N .
  eq 0 <= N = true .
  eq s(M) <= 0 = false .
  eq s(M) <= s(N) = M <= N .
  eq M > N = not (M <= N) .
endfm
```

Another example of importation is the following, where we start with the module `BASIC-NAT` in Section 2.4 and add equations in order to obtain natural numbers *modulo 3*. Here, instead of `protecting`, we use the alternative importation mechanism `including` (abbreviated to `inc`).

```
fmod NAT3 is
  including BASIC-NAT .
  var N : Nat .
  eq s(s(s(N))) = N .
endfm
```

3.2 Including vs protecting

The difference between both importation mechanisms, `protecting` and `including`, is of a semantic character.

A `protecting` importation does not change the data belonging to the sorts in the initial algebra of the imported module: The new operations do not add new values (*no junk*) to imported sorts, and the new equations do not identify old values (*no confusion*) in those sorts. Note that this is a *semantic* requirement, as opposed to a syntactic property.

In the `NAT+OPS` example, the new arithmetic operations (multiplication and subtraction) create new terms of the imported sort `Nat` and the comparison operations create new terms of sort `Bool`, but the given equations identify all these new terms with old values. Moreover, they do not identify old values because all the new equations involve in their lefthand sides new operations applied to different terms obtained by structural induction on constructors.

On the other hand, an `including` importation does not impose any semantic requirement: It is possible to add new values to the imported sorts, as well as to identify existing values in those sorts.

In the `NAT3` example, there are no new operations, and therefore no new terms are created (no junk). However, there is confusion, since the new equation identifies old terms, reducing the original initial semantics in which there is an infinite set of congruence classes (one for each natural number) to a set of just three congruence classes. Thus, this importation does not satisfy the `protecting` requirement.

Note that it is the user's responsibility to make sure that the use of `protecting` instead of `including` is correct, since this involves proving a requirement that is outside the system.

If the user wants to define natural numbers modulo 3 using natural numbers, but making sure of not destroying them, then one could use a renaming to create a new sort, say `Nat3` different from `Nat`, as explained in [13] and further illustrated by an example in Section 7.4.

3.3 Integers

By default, Maude modules implicitly import a predefined `BOOL` module. Therefore, in the remaining examples we feel free to use Boolean values and operations whenever necessary, without making such importation explicit. In addition, this imported predefined module provides very useful generic operations such as `_==_` (semantic equality checked by equational simplification), its negation `_=/=_`, and a conditional `if_then_else_fi`.

Although from the mathematical point of view we define a conditional equation as having a set of equations in the condition, note that by means of the built-in operation `_==_`, a condition of the form $u = v$ can be expressed as the Boolean term $u == v$, and, moreover, several such terms can be combined using Boolean operations. In this way, assuming that our specification is confluent and terminating, a conditional equation with several equations in the condition

$$(\overline{x} : \overline{s})\, l = r \ \ \textit{if} \ \ u_1 = v_1, \ldots, u_n = v_n$$

is equivalent to a conditional equation with a single condition

$$(\overline{x} : \overline{s})\, l = r \ \ \textit{if} \ \ (u_1 == v_1 \ \ \textit{and} \ldots \textit{and} \ \ u_n == v_n) = \textit{true}.$$

In Maude, we can abbreviate a condition of the form $b = \textit{true}$ to its lefthand side b. Therefore, conditional equations in an equational specification are usually written in the form $l = r \ \ \textit{if} \ \ b$ where b is a Boolean *term*. This is shown in the following example, specifying integers and some operations.

In this specification there are three constructors for integers: a constant `0`, a successor operation to generate positive numbers, and a predecessor operation to generate negative numbers, all of them with the corresponding attribute. Note that these three constructors are not free, since successor and predecessor are inverse of each other. In addition to the two equations expressing this property, there are several equations defining the other operations by structural induction. Note in particular the conditional equations used to define the comparison predicate `_<=_` on integers.

```
fmod INT is
  sort Int .
  op 0 : -> Int [ctor] .
  op s : Int -> Int [ctor] .
  op p : Int -> Int [ctor] .
  op _+_ : Int Int -> Int .
  op _-_ : Int Int -> Int .
  op _*_ : Int Int -> Int .
  op -_ : Int -> Int .
  op _<=_ : Int Int -> Bool .

  vars N M : Int .
  eq s(p(N)) = N .
  eq p(s(N)) = N .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
  eq p(M) + N = p(M + N) .
  eq N - 0 = N .
  eq M - s(N) = p(M - N) .
  eq M - p(N) = s(M - N) .
  eq 0 * N = 0 .
  eq s(M) * N = (M * N) + N .
  eq p(M) * N = (M * N) - N .
  eq s(M) <= N = M <= p(N) .
  eq p(M) <= N = M <= s(N) .
  eq 0 <= 0 = true .
  eq 0 <= p(0) = false .
  ceq 0 <= s(M) = true if 0 <= M .
  ceq 0 <= p(M) = false if not(0 <= M) .
endfm
```

With this more complex specification, we show the results of some equational simplifications in Maude:

```
Maude> red s(s(s(0))) * s(s(0)) <= s(s(s(0)) - s(0)) .
result Bool: false

Maude> red s(s(s(0))) * p(p(0)) <= s(p(s(0)) - s(0)) .
result Bool: true
```


To see in more detail the process of equational simplification, step by step, we can trace the rewriting process. Furthermore, the following trace has been extended by hand to highlight the subterm being rewritten.

```
Maude> set trace on .
Maude> red p(p(p(s(s(s(0)))))) <= s(p(s(0))) .

reduce in INT : p(p(p(s(s(s(0)))))) <= s(p(s(0))) .
*********** equation
eq p(s(N)) = N .
N:Int --> s(s(0))
p(s(s(s(0))))
--->
s(s(0))
---------------------------------------- p(p(p(s(s(s(0)))))) <= s(p(s(0)))
----------------------------------------     -------------
---------------------------------------- p(p(   s(s(0))   )) <= s(p(s(0)))
*********** equation
eq p(s(N)) = N .
N:Int --> s(0)
p(s(s(0)))
--->
s(0)
---------------------------------------- p(p(s(s(0)))) <= s(p(s(0)))
----------------------------------------   ----------
---------------------------------------- p(   s(0)   ) <= s(p(s(0)))
*********** equation
eq p(s(N)) = N .
N:Int --> 0
p(s(0))
--->
0
---------------------------------------- p(s(0)) <= s(p(s(0)))
---------------------------------------- -------
----------------------------------------    0    <= s(p(s(0)))
*********** equation
eq p(s(N)) = N .
N:Int --> 0
p(s(0))
--->
0
---------------------------------------- 0 <= s(p(s(0)))
----------------------------------------        -------
---------------------------------------- 0 <= s(   0   )
```

```
*********** trial #1
ceq 0 <= s(M) = true if 0 <= M = true .
M:Int --> 0
*********** equation
eq 0 <= 0 = true .
empty substitution
0 <= 0
--->
true
*********** success #1
*********** equation
ceq 0 <= s(M) = true if 0 <= M = true .
M:Int --> 0
0 <= s(0)
--->
true
---------------------------------------- 0 <= s(0)
---------------------------------------- ---------
----------------------------------------   true
result Bool: true
Maude> set trace off .
```

4 Order-sorted equational specifications

All the operations that we have used in the previous examples are totally defined. There are however arithmetic operations on natural numbers that are not defined for some values, like division, which is undefined for zero as divisor. We can often avoid the possibility of considering partial functions by extending the many-sorted equational logic to *order-sorted* equational logic, which allows us to define subsorts corresponding to the domain of definition of a function, whenever such subsorts can be defined by means of constructors.

An order-sorted signature adds a partial order relation to the set of sorts S, such that $s \leq s'$ is interpreted semantically by the subset inclusion $A_s \subseteq A_{s'}$ between the corresponding carrier sets in the algebras.

Moreover, operations can be overloaded. For example, we can use the same symbol `_+_` for addition on natural numbers and on integers inside the same specification; assuming `Nat < Int` this illustrates the notion of *subsort overloading*. Even more, the same symbol can be used in unrelated sorts, in an *ad-hoc overloading* way; for example `_+_` could be used also for disjunction on Boolean values.

Due to the subsort relation and operation overloading, a term can have several different sorts. Some requirements of monotonicity and regularity are usually imposed on order-sorted signatures to avoid bad ambiguities and ensure that each

term has a least sort. Moreover, coherence conditions on the subsort relation are necessary for equational deduction to be well-behaved in this framework. Finally, a new requirement on equations for rewriting is sort-decreasingness, that is, the least sort of the righthand side of each equation is not bigger than the least sort of its lefthand side, in addition to confluence and termination (and no new variables in the righthand side). In this context a set of equations is Church-Rosser when it is both confluent and sort-decreasing.

For more details on order-sorted equational specifications, their algebraic and operational semantics, and their relationship with many-sorted equational specifications, we refer the reader to the papers [16, 18, 24].

4.1 Natural numbers division

The successor operation on natural numbers can be used as a constructor to generate nonzero natural numbers. In this way, we can define a subsort `NzNat` of `Nat`, which is precisely the domain of definition for a division operation. Note the declaration `subsort NzNat < Nat` and the arities for the operations `_div_` and `_mod_`. Moreover, these two operations are not defined by structural induction; instead a case analysis comparing the values of their two arguments is made in the conditions of the corresponding equations.

```
fmod NAT-DIV is
  sorts Nat NzNat .
  subsort NzNat < Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> NzNat [ctor] .
  op _+_ : Nat Nat -> Nat .
  op _*_ : Nat Nat -> Nat .
  op _-_ : Nat Nat -> Nat .
  op _<=_ : Nat Nat -> Bool .
  op _>_ : Nat Nat -> Bool .
  op _div_ : Nat NzNat -> Nat .
  op _mod_ : Nat NzNat -> Nat .

  vars M N : Nat .
  var P : NzNat .

  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
  eq 0 * N = 0 .
  eq s(M) * N = (M * N) + N .
  eq 0 - N = 0 .
  eq s(M) - 0 = s(M) .
```

```
  eq s(M) - s(N) = M - N .
  eq 0 <= N = true .
  eq s(M) <= 0 = false .
  eq s(M) <= s(N) = M <= N .
  eq N > M = not (N <= M) .
  ceq N div P = 0 if P > N .
  ceq N div P = s((N - P) div P) if P <= N .
  ceq N mod P = N if P > N .
  ceq N mod P = (N - P) mod P if P <= N .
endfm

Maude > red (s(s(s(s(s(0))))) * s(s(s(s(0))))) div s(s(s(0))) .
result NzNat: s(s(s(s(s(s(0))))))
```

The reduction shows that `(5 * 4) div 3` reduces to `6` in Peano notation.

4.2 The natural numbers

In this example we simply extend the previous module of natural numbers in order to have a kind of *standard* natural numbers module with all interesting operations for possible uses in future examples. Note the use of the built-in operations `_=/=_` and `if_then_else_fi`.

```
fmod NAT is
  protecting NAT-DIV .
  op min : Nat Nat -> Nat .
  op max : Nat Nat -> Nat .
  op _<_ : Nat Nat -> Bool .
  op even? : Nat -> Bool .
  op odd? : Nat -> Bool .

  vars M N : Nat .

  eq min(N, M) = if M > N then N else M fi .
  eq max(N, M) = if M > N then M else N fi .
  eq N < M = (N <= M) and (N =/= M) .
  eq even?(0) = true .
  eq even?(s(M)) = odd?(M) .
  eq odd?(0) = false .
  eq odd?(s(M)) = even?(M) .
endfm
```


4.3 Lists of natural numbers

The same partiality question applies to the operations `head` and `tail` on lists, for example, of natural numbers, and we solve it by means of a subsort `NeList` of nonempty lists. In this example, lists are generated by the two standard free constructors: the empty list *nil*, denoted `[]`, and the *cons* operation that adds an element to the beginning of a list, denoted with the mixfix syntax `_:_`. As usual, `head` and `tail` are the selectors associated to this constructor.

The remaining operations on lists (defined as usual by structural induction on the two constructors) concatenate two lists, calculate the length of a list, reverse a list, and decompose a list into the first n elements and the rest. Finally, there is an operation to build a list consisting of all natural numbers between two given ones.

```
fmod NAT-LIST is
  protecting NAT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> List [ctor] .              *** empty list
  op _:_ : Nat List -> NeList [ctor] . *** cons
  op tail : NeList -> List .
  op head : NeList -> Nat .
  op _++_ : List List -> List .         *** concatenation
  op length : List -> Nat .
  op reverse : List -> List .
  op take_from_ : Nat List -> List .
  op throw_from_ : Nat List -> List .
  op from_to_ : Nat Nat -> List .

  vars N M : Nat .
  vars L L' : List .

  eq tail(N : L) = L .
  eq head(N : L) = N .
  eq [] ++ L = L .
  eq (N : L) ++ L' = N : (L ++ L') .
  eq length([]) = 0 .
  eq length(N : L) = s(0) + length(L) .
  eq reverse([]) = [] .
  eq reverse(N : L) = reverse(L) ++ (N : []) .

  eq take 0 from L = [] .
  eq take N from [] = [] .
```

```
  eq take s(N) from (M : L) = M : take N from L .
  eq throw 0 from L = L .
  eq throw N from [] = [] .
  eq throw s(N) from (M : L) = throw N from L  .
  ceq from N to M = [] if N > M .
  ceq from N to M = N : from s(N) to M if N <= M .
endfm
```

Usually, the possible sets of constructors are not unique. For example, as we will see in Section 5.3, it is also possible to generate lists using a different set of constructors, including the concatenation operation `_++_`. However, in this example concatenation is not a constructor, but a defined operation. It satisfies an associativity equation in the initial semantics, but not in the general loose semantics, and we have chosen not to include in this specification any equation asserting this property.

4.4 Binary trees with natural numbers

Following the same pattern of ideas as in the case of lists, we can specify binary trees with natural numbers in all nodes. They have two free constructors: the empty tree, denoted `empty-tree`, and an operation that puts a number as root below two given trees, its left and right children, denoted `_[_]_`. The three selectors associated to this constructor only make sense for nonempty trees, that belong to the corresponding subsort.

In addition to the operation that calculates the depth of a tree, we have four operations with `List` as value sort. Since all of them have the same rank, they are declared together by means of the keyword `ops`. The first returns the list of natural numbers in the leaves (or nodes in the fringe) of a tree (note the use of variables of sort `NeBinTree` in the corresponding equations to know that a node is not a leaf), and the remaining three operations are the standard binary tree traversals.

```
fmod NAT-BIN-TREE is
  protecting NAT-LIST .

  sorts NeBinTree BinTree .
  subsort NeBinTree < BinTree .

  op empty-tree : -> BinTree [ctor] .
  op _[_]_ : BinTree Nat BinTree -> NeBinTree [ctor] .
  ops left right : NeBinTree -> BinTree .
  op root : NeBinTree -> Nat .
  op depth : BinTree -> Nat .
  ops leaves preorder inorder postorder : BinTree -> List .
```



```
  vars N M : Nat .
  vars L R : BinTree .
  vars NEL NER : NeBinTree .

  eq left(L [N] R) = L .
  eq right(L [N] R) = R .
  eq root(L [N] R) = N .
  eq depth(empty-tree) = 0 .
  eq depth(L [N] R) = s(0) + max(depth(L), depth(R)) .

  eq leaves(empty-tree) = [] .
  eq leaves(empty-tree [N] empty-tree) = N : [] .
  eq leaves(NEL [N] R) = leaves(NEL) ++ leaves(R) .
  eq leaves(L [N] NER) = leaves(L) ++ leaves(NER) .

  eq preorder(empty-tree) = [] .
  eq preorder(L [N] R) = N : (preorder(L) ++ preorder(R)) .
  eq inorder(empty-tree) = [] .
  eq inorder(L [N] R) = inorder(L) ++ (N : inorder(R)) .
  eq postorder(empty-tree) = [] .
  eq postorder(L [N] R) = postorder(L)  ++ (postorder(R) ++ (N : [])) .
endfm

***       5
***    /     \
***   1       7
***  / \     / \
*** 0   3   6  11
***    / \    /  \
***   2   4  9   12
***         / \
***        8  10

Maude> red inorder(
       ((empty-tree [0] empty-tree)
        [s(0)]
        ((empty-tree [s(s(0))] empty-tree)
         [s(s(s(0)))]
         (empty-tree [s(s(s(s(0))))] empty-tree)))
       [s(s(s(s(s(0)))))]
       ((empty-tree [s(s(s(s(s(s(0))))))] empty-tree)
        [s(s(s(s(s(s(s(0)))))))]
        (((empty-tree [s(s(s(s(s(s(s(s(0))))))))] empty-tree)
```

```
          [s(s(s(s(s(s(s(s(s(0))))))))))]
          (empty-tree [s(s(s(s(s(s(s(s(s(s(0))))))))))] empty-tree))
         [s(s(s(s(s(s(s(s(s(s(s(0)))))))))))]
         (empty-tree [s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))] empty-tree)))) .

result NeList: 0 : s(0) : s(s(0)) : s(s(s(0))) : s(s(s(s(0)))) :
    s(s(s(s(s(0))))) : s(s(s(s(s(s(0)))))) : s(s(s(s(s(s(s(0))))))) :
    s(s(s(s(s(s(s(s(0)))))))) : s(s(s(s(s(s(s(s(s(0))))))))) :
    s(s(s(s(s(s(s(s(s(s(0)))))))))) : s(s(s(s(s(s(s(s(s(s(s(0))))))))))) :
    s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))) : []
*** 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10 : 11 : 12

Maude> red leaves(
       ((empty-tree [0] empty-tree)
        [s(0)]
        ((empty-tree [s(s(0))] empty-tree)
         [s(s(s(0)))]
         (empty-tree [s(s(s(s(0))))] empty-tree)))
       [s(s(s(s(s(0)))))]
       ((empty-tree [s(s(s(s(s(s(0))))))] empty-tree)
        [s(s(s(s(s(s(s(0)))))))]
        (((empty-tree [s(s(s(s(s(s(s(s(0))))))))] empty-tree)
          [s(s(s(s(s(s(s(s(s(0))))))))))]
          (empty-tree [s(s(s(s(s(s(s(s(s(s(0))))))))))] empty-tree))
         [s(s(s(s(s(s(s(s(s(s(s(0)))))))))))]
         (empty-tree [s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))] empty-tree)))) .

result NeList: 0 : s(s(0)) : s(s(s(s(0)))) : s(s(s(s(s(s(0)))))) :
    s(s(s(s(s(s(s(s(0)))))))) : s(s(s(s(s(s(s(s(s(s(0)))))))))) :
    s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))) : []
*** 0 : 2 : 4 : 6 : 8 : 10 : 12
```

5 Specifications with equational attributes

In this section we present a solution to the use of nonterminating equations like commutativity. The idea is to consider congruence classes of terms *modulo* such properties, and define rewriting, confluence, and termination on such congruence classes instead than on purely syntactic terms.

5.1 Equational attributes

In Maude, a binary operation f can be declared to satisfy some *equational axioms* by means of the attributes `assoc`, `comm`, `left id`, `right id`, and `id`. At present, such axioms include any combination of: associativity (1), commutativity (2), and left (3), right (4), or two-sided (5) identity (with respect to an identity element e).

$$\begin{array}{ll} (1) & f(x, f(y, z)) = f(f(x, y), x) \\ (2) & \qquad f(x, y) = f(y, x) \\ (3) & \qquad\quad f(e, x) = x \\ (4) & \qquad\quad f(x, e) = x \\ (5) & \quad f(e, x) = x, \quad f(x, e) = x \end{array}$$

The intended meaning of those declarations is to divide the equational theory of a functional module (or the equational part of a system module, as described later in Section 8.1) into a disjoint union $E \cup A$, with E the (conditional) equations in the module, and with $A = \cup_i A_{f_i}$ the union of all equational axioms declared in attributes for different binary operations f_i. We can then decompose the passage $t \mapsto [t]_{E\cup A}$ to congruence classes given by the unique quotient homomorphism

$$q_{E\cup A} : \mathcal{T}_\Sigma \longrightarrow \mathcal{T}_{\Sigma,E\cup A}$$

by first forming the quotient with the axioms A only, and then forming the quotient with both E and A, that is, $t \mapsto [t]_A \mapsto [t]_{E\cup A}$, given by the commutative diagram

$$\begin{tikzcd} \mathcal{T}_\Sigma \arrow[rr, "q_{E\cup A}"] \arrow[dr, "q_A"'] & & \mathcal{T}_{\Sigma,E\cup A} \\ & \mathcal{T}_{\Sigma,A} \arrow[ur, "q_{E/A}"'] & \end{tikzcd}$$

Note that we have a one-to-one correspondence between congruence classes $[t]_{E\cup A}$ in $\mathcal{T}_{\Sigma,E\cup A}$ and congruence classes $[[t]_A]_E$ in the quotient $\mathcal{T}_{\Sigma,A}/E$ of $\mathcal{T}_{\Sigma,A}$ by the equations E, that is, we have an isomorphism $\mathcal{T}_{\Sigma,A}/E \equiv \mathcal{T}_{\Sigma,E\cup A}$.

The point is that $\mathcal{T}_{\Sigma,A}$ is more abstract than $\mathcal{T}_\Sigma$, and allows us to represent a variety of commonly occurring *data structures* such as lists, multisets, and so on in a direct way, independently of their different term representations. For example, if we have the declarations

```
sort String .
ops a b c d e f g : -> String .
op _._ : String String -> String [assoc] .
```

the flattened string `a . b . c . d`, where parentheses are no longer used because they are unnecessary due to associativity, denotes exactly the congruence class modulo associativity whose members are the different parenthesized terms

```
((a . b) . c) . d,  (a . (b . c)) . d,  a . (b. (c . d)),
a . ((b . c) . d),  (a . b) . (c . d)
```

Note that the number of parenthesizations is exponential with respect to the number of elements in the string (more precisely, for a string of length n, the number of parenthesizations is the Catalan number $C(n) = \frac{1}{n}\binom{2n-2}{n-1}$). Therefore, it is not only a more convenient syntactic representation, but also a very compact representation of an exponential number of equivalent terms in the congruence class.

In general, this flattening is also applied to associative binary operations in prefix form. For example, for such an associative binary operation f, the expression $f(a, b, c)$ denotes the congruence class $\{f(a, f(b, c)), f(f(a, b), c)\}$, and this generalizes to $f(a_1, \ldots, a_n)$ for any number of arguments $n \geq 2$.

The equations E in a functional module are then assumed to be Church-Rosser (that is, confluent and sort-decreasing) and terminating *modulo* the axioms A. This means that rewriting and simplification with the equations E now takes place in A-congruence classes. For example, in the context of our previous declarations, if in E we have the equations

```
eq a . b = e .
eq c . d = f .
```

then we have the following equational simplification

$$\texttt{a . b . c . d} \rightarrow_E \texttt{e . c . d} \rightarrow_E \texttt{e . f}$$

corresponding to the equational simplification over congruence classes

$$[\texttt{(a . (b . c)) . d}]_{assoc} = [\texttt{(a . b) . (c . d)}]_{assoc} \rightarrow_E$$
$$[\texttt{e . (c . d)}]_{assoc} \rightarrow_E [\texttt{e . f}]_{assoc}.$$

Therefore, if the equations E are Church-Rosser and terminating modulo A, given a term t, there will be a unique A-congruence class that cannot be further simplified (in the above example $[\texttt{e . f}]_{assoc}$), serving as a *canonical representative* for the congruence class $[[t]_A]_E$. This generalizes the standard case in which $A = \emptyset$ and we simplify individual syntactic terms instead of congruence classes, and where if the equations are Church-Rosser and terminating we also reach as unique canonical representative of the congruence class $[t]_E$, namely, the normal form $t{\downarrow_E}$ that cannot be further simplified.

5.2 Overloading, connected components, and attributes

Since operations can be overloaded, we require all subsort overloaded versions of an operation to satisfy the same equational attributes. For example, consider the following module with a subsort declaration `NzNat < Nat` for nonzero natural numbers, as well as a sort `Nat3` for natural numbers modulo 3 and an inclusion operation i from `Nat3` into `Nat`.

```
fmod NUMBERS is
  sorts Nat NzNat Nat3 .
  subsort NzNat < Nat .

  op zero : -> Nat [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  op _+_ : NzNat NzNat -> NzNat [assoc comm] .

  ops 0 1 2 : -> Nat3 [ctor] .
  op _+_ : Nat3 Nat3 -> Nat3 [comm] .
  op i : Nat3 -> Nat .

  vars N M : Nat .
  var N3 : Nat3 .

  eq N + zero = N .
  eq N + s M = s (N + M) .
  eq N3 + 0 = N3 .
  eq 1 + 1 = 2 .
  eq 1 + 2 = 0 .
  eq 2 + 2 = 1 .
  eq i(0) = zero .
  eq i(1) = s zero .
  eq i(2) = s s zero .
endfm
```

The addition operation `_+_` has been overloaded in two ways. First, it is subsort overloaded over `NzNat` and `Nat`, because these two sorts are related in the subsort relation, that is, they belong to the same connected component in the partial order of sorts under the subsort relation. Then, both declarations must have exactly the same equational attributes `assoc` and `comm`.

Second, there is also an ad-hoc overloaded declaration of `_+_` over `Nat3` that belongs to a different connected component in the subsort relation. In this situation, the set of equational attributes for this declaration may be different, for example, having only `comm` instead of both `assoc` and `comm` in the declaration above.

The general requirement is that subsort overloaded operations must all have the same equational attributes, but this is not required for ad-hoc overloaded operations. This assumption is used by the Maude interpreter to use the same internal representation and matching algorithms for terms built with the same subsort overloaded operation.

5.3 Associativity: Lists

We are going to specify lists of natural numbers using a different set of constructors (with respect to the set of constructors used in Section 4.3), making use of equational attributes. We no longer have a cons operation `_:_`, and instead use concatenation as the main list constructor (written now using empty juxtaposition notation) based on singleton lists, which are identified with the basic elements (natural numbers in this case) via the declaration of `Nat` as subsort of `List`. However, concatenation cannot be a free list constructor, because it satisfies an associativity equation. This equation is not declared as such, but as an attribute of the operation. Moreover, there is also an identity relationship of concatenation with respect to the empty list `nil`, which is expressed with two equations (but see also the example in Section 5.4).

Notice how the concatenation operation is subsort overloaded, having one declaration for nonempty lists and another one for lists, both with the same `assoc` attribute. There are two more possibilities of concatenation overloading (`NeList List -> NeList` and `List NeList -> NeList`) but they are unnecessary in this case because of the equations for identity.

```
fmod NAT-LIST-ASSOC is
  protecting NAT .

  sorts NeList List .
  subsorts Nat < NeList < List .

  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc] .
  op __ : NeList NeList -> NeList [ctor assoc] .

  op tail : NeList -> List .
  op head : NeList -> Nat .
  op length : List -> Nat .
  op reverse : List -> List .

  var N : Nat .
  var L : List .
  var L' : NeList .
```



```
  eq nil L = L .
  eq L nil = L .
  eq tail(N L) = L .
  eq head(N L) = N .
  eq length(nil) = 0 .
  eq length(N) = s(0) .
  eq length(N L') = s(0) + length(L') .
  eq reverse(nil) = nil .
  eq reverse(N) = N .
  eq reverse(N L') = reverse(L') N .
endfm
```

Instead of giving all the operations of the specification `NAT-LIST` in Section 4.3, in this specification we only have the operations `length` and `reverse`, defined as usual by structural induction on constructors. Notice that for each of the two operations there are three equations, corresponding to the empty list, singleton lists identified with natural numbers, and lists with at least two elements.

5.4 Associativity + Identity: Lists

We consider a small variant of the previous example where concatenation is the main constructor for lists, declared this time with both an attribute `assoc` for associativity, and an attribute `id: nil` for the empty list as two-sided identity.

The important point to note is that with identity as an attribute the equations for identity belong to the set A as discussed in Section 5.1. In this way, congruence classes over which rewriting and simplification take place are calculated modulo associativity and (two-sided) identity. Therefore, we only need two equations to completely specify the behavior of defined operations like `length` and `reverse`; the singleton case becomes a particular case of the `N L` case by instantiating the variable `L` with the constant `nil` and applying the identity equation.

```
fmod NAT-LIST-ASSOC-ID is
  protecting NAT .

  sorts NeList List .
  subsorts Nat < NeList < List .

  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
  op __ : NeList NeList -> NeList [ctor assoc id: nil] .

  op tail : NeList -> List .
  op head : NeList -> Nat .
```

```
  op length : List -> Nat .
  op reverse : List -> List .

  var N : Nat .
  var L : List .

  eq tail(N L) = L .
  eq head(N L) = N .
  eq length(nil) = 0 .
  eq length(N L) = s(0) + length(L) .
  eq reverse(nil) = nil .
  eq reverse(N L) = reverse(L) N .
endfm
```

There is another thing to watch out for in the presence of the identity attribute: The alternative equation `length(L L') = length(L) + length(L')` (with `L` and `L'` variables of sort `List`) causes problems of nontermination. To see this, consider the instantiation with `L'`$\mapsto$ `nil` that gives

```
  length(L nil) = length(L) + length(nil) = length(L nil) + length(nil) =
  (length(L) + length(nil)) + length(nil) = ...
```

because of the identification `L = L nil`.

Finally, notice that attributes in overloaded operations have to coincide as described in Section 5.2, even though, by reading alone the second declaration for concatenation, it may sound a bit strange to say that the empty list is an identity for an operation only defined on nonempty lists.

5.5 Associativity + Commutativity + Identity: Multisets

In the same way as associativity and identity for concatenation provide structural axioms for lists (or strings), we can specify abstractly multisets (for example, of natural numbers) by considering a union constructor (written again with empty juxtaposition syntax) that satisfies associativity, commutativity (because now order between elements does not matter), and identity structural axioms, all declared as attributes.

```
fmod NAT-MSET is
  protecting NAT .
  sort Mset .
  subsorts Nat < Mset .
  op empty-mset : -> Mset [ctor] .
  op __ : Mset Mset -> Mset [ctor assoc comm id: empty-mset] .
```



```
  op size : Mset -> Nat .
  op mult : Nat Mset -> Nat .
  op delete : Nat Mset -> Mset .
  op _in_ : Nat Mset -> Bool .

  vars N N' : Nat .
  var S : Mset .

  eq size(empty-mset) = 0 .
  eq size(N S) = s(0) + size(S) .
  eq mult(N, empty-mset) = 0 .
  eq mult(N, N S) = s(0) + mult(N, S) .
  ceq mult(N, N' S) = mult(N, S) if N =/= N' .
  eq delete(N, empty-mset) = empty-mset .
  eq delete(N, N S) = delete(N, S) .
  ceq delete(N, N' S) = N' delete(N, S) if N =/= N' .
  eq N in S = mult(N, S) > 0 .
endfm
```

Notice again the form of the equations for the defined operations; for example, an equation like `size(S S') = size(S) + size(S')` (with `S` and `S'` variables of sort `Mset`) would cause problems of nontermination in the presence of the identity attribute.

5.6 Associativity + Commutativity + Identity + Idempotency: Sets

Building on the multiset example of the previous section, we can get a specification for sets, where multiplicity of elements does not matter, by adding idempotency as an equation.

```
fmod NAT-SET is
  protecting NAT .
  sort Set .
  subsorts Nat < Set .
  op empty-set : -> Set [ctor] .
  op __ : Set Set -> Set [ctor assoc comm id: empty-set] .

  op _in_ : Nat Set -> Bool .
  op delete : Nat Set -> Set .
  op card : Set -> Nat .
  op _-_ : Set Set -> Set .
```

```
  vars N N' : Nat .
  vars S S' : Set .

  eq N N = N .
  eq N in empty-set = false .
  eq N in (N' S) = (N == N') or (N in S) .
  eq delete(N, empty-set) = empty-set .
  eq delete(N, N S) = delete(N, S) .
  ceq delete(N, N' S) = N' delete(N, S) if N =/= N' .
  eq S - empty-set = S .
  eq S - (N S') = delete(N, S) - S' .
  eq card(empty-set) = 0 .
  eq card(N S) = s(0) + card(delete(N,S)) .
endfm
```

Notice that the equations for the operations `delete` and `card` above make sure on the righthand side that further occurrences of `N` in `S` are also deleted or not counted, respectively, because we cannot rely on the order in which equations are applied in equational simplification to assume that there are no repeated elements in an expression.

Notice also that the idempotency equation is stated only for singleton sets. Stating it for arbitrary sets in the form `S S = S` would cause nontermination because of the identity attribute:

$$\texttt{empty-set} = \texttt{emptyset emptyset} \to \texttt{empty-set}\ldots$$

5.7 Idempotent semigroups

This example is of a different character, as opposed to the data structure character of the previous examples in this section.

Consider the word problem for *idempotent semigroups*: Given a binary concatenation operation `__` satisfying the equations for associativity $(xy)z = x(yz)$ and idempotency $xx = x$, when are two words provably equal? For example, do we have $abc = abcbabc$? (Note the absence of parentheses because of associativity.)

The first idea is to build in the associative equation by means of an equational attribute, and use the idempotency equation from left to right as a rewrite rule.

The imported module `QID` is a useful built-in module providing quoted identifiers that in this case represent the generators of the semigroup.

```
fmod NAIVE-ASSOC-IDEMP is
  protecting QID .
  sort List .
```



```
  subsort Qid < List .
  op __ : List List -> List [ctor assoc] . *** list concatenation
  var L : List .
  eq L L = L .
endfm
```

When we query the Maude system for the equality mentioned above, we obtain

```
Maude> red 'a 'b 'c == 'a 'b 'c 'b 'a 'b 'c .
result Bool: false
```

This result is *wrong*, because these two terms must be identified in the equational theory since both of them can be proved equal to `'a 'b 'a 'b 'c 'b 'a 'b 'c` as follows:

$$\begin{tikzcd}
 & \texttt{'a 'b 'a 'b 'c 'b 'a 'b 'c} \arrow[dl] \arrow[ddr] & \\
\texttt{'a 'b 'a 'b 'c} \arrow[d] & & \\
\texttt{'a 'b 'c} & & \texttt{'a 'b 'c 'b 'a 'b 'c}
\end{tikzcd}$$

The problem is lack of *confluence*. The above specification can be made confluent (while preserving termination, which is obvious, since the length of the word is strictly shorter after each equational simplification) by adding one conditional rule [28]. This rule is quite subtle, because its condition involves comparing *sets* of letters in subwords. Note in the following specification that both lists and sets are nonempty, because there is no use for the empty case, and this at the same time avoids nontermination problems with some equations as we have already mentioned before.

```
fmod ASSOC-IDEMP is
  protecting QID .
  sorts List Set .
  subsorts Qid < List Set .
  op __ : List List -> List [ctor assoc] .    *** list concatenation
  op _,_ : Set Set -> Set [ctor assoc comm] . *** set union
  op {_} : List -> Set .                      *** set of a list

  var I : Qid .
  var S : Set .
  vars L P Q : List .

  eq S,S = S .
  eq {I} = I .
```

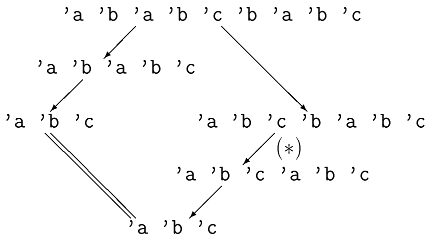
```
  eq {I L} = I,{L} .
  eq L L = L .
  ceq L P Q = L Q if {L} == {Q} and {L P} == {L} .
endfm

Maude> red 'a 'b 'c .
result List: 'a 'b 'c

Maude> red 'a 'b 'c 'b 'a 'b 'c .
result List: 'a 'b 'c

Maude> red 'a 'b 'c == 'a 'b 'c 'b 'a 'b 'c .
result Bool: true
```

Now we have obtained the correct result because the specification is indeed confluent; the example in the diagram above can be completed by using the conditional equation in the step (∗).



5.8 More on matching and rewriting modulo

In the Maude implementation, rewriting modulo A is accomplished by using a *matching modulo A algorithm*. More precisely, given an equational theory A, a term t (corresponding to the lefthand side of an equation) and a subject term u, we say that *t matches u modulo A* (or that *t A-matches u*) if there is a substitution σ such that $\sigma(t) =_A u$, that is, $\sigma(t)$ and u are equal modulo the equational theory A (compare with the syntactic definition of matching in Section 2.2).

Given an equational theory $A = \cup_i A_{f_i}$ corresponding to all the attributes declared in different binary operations, Maude synthesizes a combined matching algorithm for the theory A, and does both equational simplification (with equations) and rewriting (with rules in system modules, see Section 8.1) modulo the axioms A.

Note, however, that for operations f whose equational axioms A include the associativity axiom, to achieve the effect of simplification modulo A using an A-matching algorithm, we have to attempt matching a lefthand side of the form $f(t_1, t_2)$ not only

on a subject term u, but also on all its f-subterms, that is, on those "fragments" of the top structure of the term that could be matched by $f(t_1, t_2)$. For example, for $f =$ `_._` and $u =$ `a . b . c . d` (as in Section 5.1), the lefthand side of the equation `a . b = e` does not match the string u (that is, there is no substitution making both strings equal modulo associativity), but it trivially matches the fragment `a . b`. For the case where the only axiom is associativity, the `_._`-subterms of `a . b . c . d` are

```
a . b,  a . b . c,  b . c,  b . c . d,  c . d
```

If the operation `_._` had been declared both associative and commutative, then we should add to those the additional subterms

```
a . c,  a . d,  b . d,  a . b . d,  a . c . d
```

If the term $f(t_1, t_2)$ matches either u or an f-subterm of u modulo A, then we say that *$f(t_1, t_2)$ matches u with extension modulo A* (or that *$f(t_1, t_2)$ A-matches u with extension*). For example, the lefthand side of the equation `a . b = e` matches `a . b . c . d` with extension modulo associativity, and the lefthand side of `a . d = g` matches `a . b . c . d` with extension modulo associativity and commutativity.

For f a binary function symbol with equational attributes A_f including the associativity axiom, we now define how a subject term u is *A_f-rewritten with extension* using an equation $f(t_1, t_2) = v$. First of all, $f(t_1, t_2)$ must A_f-match with extension a *maximal* f-subterm w of u (that is, an f-subterm of u that is not itself an f-subterm of a bigger f-subterm). This means that there is an f-subterm w_0 of w and a substitution σ such that $\sigma(f(t_1, t_2)) =_{A_f} w_0$. Then, the corresponding A_f-rewriting with extension step rewrites u to the term obtained by replacing the subterm w_0 by $\sigma(v)$.

Note that a term $f(t_1, t_2)$ A_f-matches with extension a maximal f-subterm if and only if it A_f-matches without extension *some* f-subterm. This is of course the important practical advantage of A-matching and A-rewriting with extension, namely, that only maximal f-subterms of a term have to be inspected to get the effect of rewriting A-congruence classes. For more technical details on rewriting modulo a set of axioms, see [12].

Matching with extension for an associative operation essentially corresponds to matching without extension for a collection of associated equations. For example, we could have "generalized" the equation `a . b = e` with `_._` associative to the equations

```
eq a . b = e .
eq X . a . b = X . e .
eq a . b . Y = e . Y .
eq X . a . b . Y = X . e . Y .
```

so that we could have achieved the same effect by rewriting only at the top of maximal f-subterms (without extension). Similarly, for `_._` associative and commutative, we could have generalized the same equations to the equations

```
eq a . b = e .
eq a . b . Y = e . Y .
```

In Maude this generalization does not have to be performed explicitly as a transformation of the specification. It is instead achieved implicitly in a built-in way by performing A-matching with extension. If the equational axioms declared for a binary operation f include the associativity axiom, then a subject term u with f as its top operation is internally represented (but this representation can also be externally used!) as the flattened term $f(u_1, \ldots, u_n)$, with the $u_1, \ldots, u_n$ having top operations different from f. Furthermore, if a (two-sided) identity element e has been declared for f, then $u_i \neq e, 1 \leq i \leq n$. That is, we assume in this case that all identities have been simplified away.

Relative to this internal representation, it is then easy to define the notion of an f-subterm. If the axioms of f include associativity but not commutativity, then the f-subterms of $f(u_1, \ldots, u_n)$ are all terms of the form $f(u_k, \ldots, u_{k+h})$ with $1 \leq k \leq n-1$ and $1 \leq h \leq n-k$.

Similarly, if the axioms of f include associativity and commutativity, then the f-subterms of $f(u_1, \ldots, u_n)$ are all terms of the form $f(u_{k_1}, \ldots, u_{k_h})$ with $1 \leq k_{i_1} < \cdots < k_{i_h} \leq n$, and $2 \leq h \leq n$.

Adding axioms for an identity element e to a possibly associative and/or commutative operation f leads to some subtle cases where the proper application of the general notions may not always coincide with the user's expectations. To begin with, as we have already illustrated in several specifications, unexpected cases of *nontermination* may be introduced by an unwary user. For example, the equation

```
eq a . X = b . a .
```

will cause nontermination when `_._` is declared associative with identity `1`, since we have, for example,

$$\texttt{d . a} \rightarrow \texttt{d . b . a} \rightarrow \texttt{d . b . b . a} \rightarrow \cdots \rightarrow \texttt{d . b}^n \texttt{ . a} \rightarrow \cdots$$

by instantiating each time the variable `X` to the identity element `1`.

A second source of unexpected behavior is the fact that a lefthand side involving an associative operation may, in the presence of an additional identity attribute, match a term not involving at all that operation. Thus, for the above equation, we have also the nonterminating chain of rewriting steps

$$\texttt{a} \rightarrow \texttt{b . a} \rightarrow \texttt{b . b . a} \rightarrow \cdots \rightarrow \texttt{b}^n \texttt{ . a} \rightarrow \cdots$$

Similar problems have already been mentioned in Section 5.4 regarding the form of the equations for the operation `length` on lists, and in Section 5.5 for the operation `size` on multisets.

In a similar way, in the presence of an identity element, the user's expectations about when a lefthand side will match with extension a subject term may not fully agree with the proper technical definition. Consider for example a binary operation `_._` that is associative and commutative, and that has an identity element `1`, and let

```
eq a . X = c .
```

be an equation. Then,

1. The lefthand side `a . X` matches the subject term `a` modulo the axioms by instantiating `X` to `1`, giving rise to the simplification $\texttt{a} \rightarrow \texttt{c}$.

2. The same lefthand side matches the subject term `a . b . c` with extension in *three* different ways, namely, with substitutions $\texttt{X} \mapsto \texttt{b . c}$, $\texttt{X} \mapsto \texttt{b}$, and $\texttt{X} \mapsto \texttt{c}$, giving rise to the three simplifications

   $$\texttt{a . b . c} \rightarrow \texttt{c}, \quad \texttt{a . b . c} \rightarrow \texttt{c . c}, \quad \texttt{a . b . c} \rightarrow \texttt{b . c}.$$

3. For the same subject term `a . b . c`, the substitution $\texttt{X} \mapsto \texttt{1}$ *is not* a match with extension of the above lefthand side, because the term `a . 1` is not a `_._`-subterm of the term `a . b . c`. However, because of 1 above, we know that the equation will match that way not at the top, but "one level down," leading to the simplification $\texttt{a . b . c} \rightarrow \texttt{c . b . c}$.

It is also important to realize that there is no match with extension between the lefthand side of the equation `a = b` and the subject term `a . b . c`, although again the equation will match that way not at the top, but "one level down," leading to the simplification $\texttt{a . b . c} \rightarrow \texttt{b . b . c}$.

Of course, lefthand sides may contain several operations, each matched modulo a different theory. Maude will then match each fragment of a lefthand side according to its given theory. Consider, for example, the following specification where `_._` is associative and `_+_` is associative and commutative:

```
fmod XMATCH-TEST is
  sort Elt .
  ops a b c d e : -> Elt .
  op _._ : Elt Elt -> Elt [assoc] .
  op _+_ : Elt Elt -> Elt [assoc comm] .
  vars X Y Z : Elt .
  eq X . (Y + Z) = (X . Y) + (X . Z) .  *** distributivity
endfm
```

The lefthand side of the distributivity equation will produce 12 matches with extension for the subject term

```
a . b . (c + d + e)
```

Enumerating these by hand would be tedious and error prone, however Maude provides the *xmatch* command for just this purpose:

```
Maude> xmatch X . (Y + Z) <=? a . b . (c + d + e) .
```

The output consists of the substitution for each match with extension together with the portion of the subject actually matched:

```
xmatch in XMATCH-TEST : X . (Y + Z) <=? a . b . (c + d + e) .
Solution 1
Matched portion = (whole)
X:Elt --> a . b
Y:Elt --> c
Z:Elt --> d + e

Solution 2
Matched portion = b . (c + d + e)
X:Elt --> b
Y:Elt --> c
Z:Elt --> d + e

Solution 3
Matched portion = (whole)
X:Elt --> a . b
Y:Elt --> d
Z:Elt --> c + e

Solution 4
Matched portion = b . (c + d + e)
X:Elt --> b
Y:Elt --> d
Z:Elt --> c + e

Solution 5
Matched portion = (whole)
X:Elt --> a . b
Y:Elt --> e
Z:Elt --> c + d
```



```
Solution 6
Matched portion = b . (c + d + e)
X:Elt --> b
Y:Elt --> e
Z:Elt --> c + d

Solution 7
Matched portion = (whole)
X:Elt --> a . b
Y:Elt --> c + d
Z:Elt --> e

Solution 8
Matched portion = b . (c + d + e)
X:Elt --> b
Y:Elt --> c + d
Z:Elt --> e

Solution 9
Matched portion = (whole)
X:Elt --> a . b
Y:Elt --> c + e
Z:Elt --> d

Solution 10
Matched portion = b . (c + d + e)
X:Elt --> b
Y:Elt --> c + e
Z:Elt --> d

Solution 11
Matched portion = (whole)
X:Elt --> a . b
Y:Elt --> d + e
Z:Elt --> c

Solution 12
Matched portion = b . (c + d + e)
X:Elt --> b
Y:Elt --> d + e
Z:Elt --> c
```

Note that extension is only used for matching the top operation, `_._` in this example, but not `_+_`. This is because the subterm `Y + Z` of the lefthand side should of course match the entire maximal +-subterm of the subject term, and not just some +-subterm.

6 Membership equational logic specifications

In order-sorted equational specifications, subsorts must be defined by means of constructors, but it is not possible to have a subsort of ordered lists, for example, defined by a property over lists; a more expressive formalism is needed.

Even within the order-sorted framework, there is a different problem of a more syntactic character. In the example of natural numbers division in Section 4.1, consider the term `s(s(s(0))) div (s(s(0)) - s(0))` which obviously should be equal to `s(s(s(0)))`, that is, in more usual notation $3/(2-1) = 3$. The problem is that the term `s(s(s(0))) div (s(s(0)) - s(0))` is not even well formed and it does not parse. The subterm `s(s(0)) - s(0)` has least sort `Nat`, while the `div` operation expects its second argument to be of sort `NzNat < Nat`, to guarantee precisely welldefinedness. Of course, this is too restrictive and makes most (really) order-sorted specifications useless, unless there is a mechanism that gives at parsing time the benefit of the doubt to this kind of terms. A possible such mechanism was based on an extension of the original specification by *retracts* [16].

A logically cleaner solution is to extend the logic in such a way that parsing takes place at the many-sorted level, and the sentences of the logic take care of subsorting and of assigning sorts to terms.

Membership equational logic solves both problems, by introducing sorts as predicates and allowing subsort definition by means of conditions involving equations and/or sort predicates.

6.1 Membership equational logic

A signature in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ where K is a set of *kinds*, (K, Σ) is a many-sorted (although it is better to say "many-kinded") signature, and $S = \{S_k\}_{k \in K}$ is a K-kinded set of *sorts*.

An Ω-*algebra* is then a (K, Σ)-algebra $\mathbf{A}$ together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$. Intuitively, the elements in sorts are the good, or correct, or non-error, or defined, elements, whereas the elements without a sort are error or undefined elements.

Atomic formulas are either Σ-equations, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. General sentences are Horn clauses on these atomic formulas, quantified by finite sets of K-kinded variables. That is, they are either conditional equations

$$(\forall X)\ \ t = t' \ \ \mathit{if}\ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

or (conditional) *membership axioms* of the form

$$(\forall X)\ \ t : s \ \ \mathit{if}\ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

Membership equational logic is equivalent to Horn clause logic with equality and has the usual good properties of soundness and completeness with respect to appropriate rules of equational deduction; the classes of algebras satisfying a specification have initial models; and moreover, membership equational logic conservatively extends both many-sorted and order-sorted equational logic [24].

In Maude, *functional modules* are membership equational specifications and their semantics is given by the corresponding *initial membership algebra* in the class of algebras satisfying the specification.

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are not declared explicitly, and intuitively correspond to the connected components of the subsort relation. The convenience of order-sorted notation is retained as syntactic sugar. Thus, an operation declaration like

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

is understood as syntactic sugar for a declaration at the kind level together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

Similarly, a subsort declaration `NzNat < Nat` corresponds to the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

Computation in a functional module is accomplished by using the equations as rewrite rules until a canonical form is found. Therefore, the equations must satisfy the additional requirements of being Church-Rosser and terminating [3]. This guarantees that all terms in a congruence class modulo the equations will rewrite to a unique canonical form, and that this canonical form can be assigned a sort that is smaller than all other sorts assignable to terms in the class. Since Maude supports rewriting modulo equational theories such as associativity or associativity/commutativity, all that we say has to be understood for equational rewriting *modulo such axioms*.

In addition to just constructors, which was all we had in order-sorted equational specifications, subsorts can now be defined by means of (conditional) membership axioms where conditions may include equations and/or sort predicates. Then, Maude uses these membership axioms to check that certain terms have appropriate sorts, in particular before applying an equation to do equational simplification. In this way, the system guarantees that computation takes place over "good" terms, that is, terms that have a sort in addition to having a kind. It is important to emphasize that parsing takes place at the kind level, but terms that fail to have a sort are considered error terms that in principle stop the computation. Although it is possible to introduce in this logical framework possibilities for error and exception recovery by means of a theory transformation, the standard operational semantics of membership equational logic specification in Maude only simplifies subterms that have a sort.

This means that equation declarations at the Maude user level also have some syntactic sugar in the sense that a declaration

```
ceq l = r if C .
```

is translated into the corresponding membership equational logic specification as follows, where `s` is the top sort in the kind of the lefthand side term `l` (if there are several maximal sorts in the kind, then there is an equation for each one).

```
ceq l = r if C and l : s .
```

Therefore, from a computational point of view, equational simplification only takes place on terms or subterms that belong to a sort.

Also, concerning the semantic requirement of *no junk* and *no confusion* for `protecting` importations, it must be clarified that in this context, it only applies to "good" terms, that is, to terms that belong to a sort, and not to error terms in kinds. This is because, since error terms are not equated, a `protecting` importation in *this* sense may typically add junk at the kind level while protecting data in sorts.

For more details on membership equational logic, its relationship with order-sorted equational specifications and partial equational specifications, and its operational semantics, we refer the reader to the papers [24, 3].

6.2 Paths

Our first example using the new features of membership equational logic specifies the set of paths over a given graph. A path is a sequence of edges satisfying the additional equational requirement that, for each pair of consecutive edges $e_1; e_2$, the target of e_1 coincides with the source of e_2.

First we define a concrete graph in the module `A-GRAPH`, by enumerating finite sets of nodes and edges, and by defining source and target functions also by simple enumeration. The graphical representation of the graph is depicted in Figure 1.

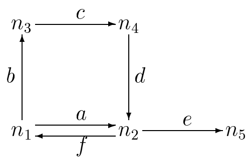


Figure 1: A graph.

```
fmod A-GRAPH is
  sorts Edge Node .
  ops n1 n2 n3 n4 n5 : -> Node [ctor] .
  ops a b c d e f : -> Edge [ctor] .
  ops source target : Edge -> Node .

  eq source(a) = n1 .  eq target(a) = n2 .
  eq source(b) = n1 .  eq target(b) = n3 .
  eq source(c) = n3 .  eq target(c) = n4 .
  eq source(d) = n4 .  eq target(d) = n2 .
  eq source(e) = n2 .  eq target(e) = n5 .
  eq source(f) = n2 .  eq target(f) = n1 .
endfm
```

Now, we specify the set of paths over the given graph by specifying a sort `Path` as an equationally defined subsort (by means of a conditional membership axiom) of a sort `Path?` that contains all sequences of edges, even those that do not satisfy the path requirement.

There are three additional operations defined exclusively on paths. The source of a path is the source of its first edge, the target of a path is the target of its last egde, and the length of path is the number of edges that constitute it.

```
fmod PATH is
  protecting NAT .
  protecting A-GRAPH .

  sorts Path Path? .
  subsorts Edge < Path < Path? .
  op _;_ : Path? Path? -> Path? [ctor assoc] .
  ops source target : Path -> Node .
  op length : Path -> Nat .

  var E : Edge .
  var P : Path .
```

```
  cmb (E ; P) : Path if target(E) == source(P) .

  eq source(E ; P) = source(E) .
  eq target(P ; E) = target(E) .
  eq length(E) = s(0) .
  eq length(E ; P) = s(0) + length(P) .
endfm
```

Note that each one of the equations above is translated in the membership equational logic semantics to a conditional equation that makes sure the lefthand side is a well-sorted term.

Some examples of reductions are the following:

```
Maude> red length(b ; c ; d) .
result NzNat: s(s(s(0)))

Maude> red length(a ; b ; c) .
result Error(Nat): length(a ; b ; c)
```

Notice the error result obtained when trying to apply the `length` operation to a term that does not represent a well-formed path.

6.3 Ordered lists

In the following example we take advantage of the additional expressiveness of membership equational logic to define an equationally defined subsort of *ordered lists* of natural numbers, which are imported from the module `NAT-LIST` in Section 4.3. Notice the three (conditional) membership axioms defining the sort `OrdList`.

In addition, we specify several classical sorting functions: `insertion-sort`, `quicksort`, and `mergesort`. Each one of them uses appropriate auxiliary operations whose behavior is the expected one; for example, `mergesort` halves a list, recursively sorts each half of the list, and then calls a `merge` operation that merges two ordered lists into an ordered list.

The important point is that we are able to give finer typing to all these sorting operations than the usual typing in other algebraic specification frameworks. Thus, `mergesort` is declared as an operation from `List` to `OrdList`, instead of the much less informative typing from `List` to `List`. The same applies to each one of the auxiliary operations. Also, a function that requires its input argument to be an ordered list can now be defined as a *total* function, whereas in less expressive typing formalisms it would have to be either *partial*, or defined with exceptional behavior on the erroneous arguments.


```
fmod NAT-ORD-LIST is
  protecting NAT-LIST .

  sorts OrdList NeOrdList .
  subsorts NeOrdList < OrdList NeList < List .

  op insertion-sort : List -> OrdList .
  op insert-list : OrdList Nat -> OrdList .

  op mergesort : List -> OrdList .
  op merge : OrdList OrdList -> OrdList .

  op quicksort : List -> OrdList .
  op leq-elems : List Nat -> List .
  op gr-elems : List Nat -> List .

  vars N M : Nat .
  vars L L' : List .
  vars OL OL' : OrdList .
  var NEOL : NeOrdList .

  mb [] : OrdList .
  mb N : [] : NeOrdList .
  cmb N : NEOL : NeOrdList if N <= head(NEOL) .

  eq insertion-sort([]) = [] .
  eq insertion-sort(N : L) = insert-list(insertion-sort(L), N) .

  eq insert-list([], M) = M : [] .
  ceq insert-list(N : OL, M) = M : N : OL if M <= N .
  ceq insert-list(N : OL, M) = N : insert-list(OL, M) if M > N .

  eq mergesort([]) = [] .
  eq mergesort(N : []) = N : [] .
  ceq mergesort(L)
     = merge(mergesort(take (length(L) div s(s(0))) from L),
             mergesort(throw (length(L) div s(s(0))) from L))
      if length(L) > s(0) .

  eq merge(OL, []) = OL .
  eq merge([], OL) = OL .
  ceq merge(N : OL, M : OL') = N : merge(OL, M : OL') if N <= M .
  ceq merge(N : OL, M : OL') = M : merge(N : OL, OL') if N > M .
```

```
  eq quicksort([]) = [] .
  eq quicksort(N : L)
    = quicksort(leq-elems(L,N)) ++ (N : quicksort(gr-elems(L,N))) .

  eq leq-elems([], M) = [] .
  ceq leq-elems(N : L, M) = N : leq-elems(L, M) if N <= M .
  ceq leq-elems(N : L, M) = leq-elems(L, M) if N > M .
  eq gr-elems([], M) = [] .
  ceq gr-elems(N : L, M) = gr-elems(L, M) if N <= M .
  ceq gr-elems(N : L, M) = N : gr-elems(L, M) if N > M .
endfm

Maude> red insertion-sort(s(s(s(s(0)))) : s(s(s(0))) : s(s(0)) :
                          s(0) : 0 : []) .
result NeOrdList: 0 : s(0) : s(s(0)) : s(s(s(0))) : s(s(s(s(0)))) : []

Maude> red mergesort(s(s(s(s(0)))) : s(s(s(0))) : s(s(0)) :
                     s(0) : 0 : []) .
result NeOrdList: 0 : s(0) : s(s(0)) : s(s(s(0))) : s(s(s(s(0)))) : []

Maude> red quicksort(s(s(s(s(0)))) : s(s(s(0))) : s(s(0)) :
                     s(0) : 0 : []) .
result NeOrdList: 0 : s(0) : s(s(0)) : s(s(s(0))) : s(s(s(s(0)))) : []
```

6.4 Binary search trees

This example is similar in philosophy to the previous one, but is a bit more complex. We specify a subsort of *(binary) search trees* by using several (conditional) membership axioms over terms of the sort `BinTree` of binary trees defined in Section 4.4.

First note that although we allowed repeated elements in an ordered list, this is not the case in a search tree, where all natural numbers must be different. As usual, a search tree is either the empty binary tree or a nonempty binary tree such that all elements in the left child are smaller than the element in the root, and all elements in the right child are bigger than it. This is checked by means of auxiliary functions that calculate the minimum and maximum element in a nonempty search tree that are also useful when deleting an element.

Typical operations on this data type include insertion of an element (if it is already in the tree, this is not altered), search of an element (with a Boolean value as result), and deletion of an element (if it is not in the tree, this is not modified either). All of them are defined by structural induction on the tree and then by a case analysis according to whether the element given as argument is smaller, equal, or bigger than the element in the root of the nonempty tree. The `delete` operation

uses the auxiliary operation `min` to create a root element from the right child when the existing root is deleted (equivalently, it could use the auxiliary operation `max` with the left child).

Again, the most important point is that membership equational logic allows us both to define the corresponding subsort and to assign typings in the best possible way to all the operations defined for this data type, without ever needing a partial operation.

```
fmod NAT-SEARCH-TREE is
  protecting NAT-BIN-TREE .

  sorts SearchTree NeSearchTree .
  subsorts NeSearchTree < SearchTree < BinTree .
  subsort NeSearchTree < NeBinTree .

  ops insert delete : SearchTree Nat -> SearchTree .
  op find : SearchTree Nat -> Bool .
  ops min max : NeSearchTree -> Nat .

  vars N M : Nat .
  vars SL SR : SearchTree .
  vars NESL NESR : NeSearchTree .

  mb empty-tree : SearchTree .
  mb empty-tree [N] empty-tree : NeSearchTree .
  cmb NESL [N] empty-tree : NeSearchTree if max(NESL) < N .
  cmb empty-tree [N] NESR : NeSearchTree if N < min(NESR) .
  cmb NESL [N] NESR : NeSearchTree if max(NESL) < N and N < min(NESR) .

  eq insert(empty-tree, M) = empty-tree [M] empty-tree .
  eq insert(SL [N] SR, N) = SL [N] SR .
  ceq insert(SL [N] SR, M) = insert(SL, M) [N] SR if M < N .
  ceq insert(SL [N] SR, M) = SL [N] insert(SR, M) if N < M .

  eq delete(empty-tree, N) = empty-tree .
  ceq delete(SL [N] SR, M) = delete(SL, M) [N] SR if M < N .
  ceq delete(SL [N] SR, M) = SL [N] delete(SR, M) if M > N .
  eq delete(empty-tree [N] SR, N) = SR .
  eq delete(SL [N] empty-tree, N) = SL .
  eq delete(NESL [N] NESR, N) = NESL [min(NESR)] delete(NESR, min(NESR)) .

  eq find(empty-tree, N) = false .
  eq find(SL [N] SR, N) = true .
  ceq find(SL [N] SR, M) = find(SL, M) if M < N .
```

```
  ceq find(SL [N] SR, M) = find(SR, M) if M > N .

  eq min(empty-tree [N] SR) = N .
  eq min(NESL [N] SR) = min(NESL) .
  eq max(SL [N] empty-tree) = N .
  eq max(SL [N] NESR) = max(NESR) .
endfm

***       5
***    /     \
***   1       7
***  / \     / \
*** 0   3   6  11
***    / \    /  \
***   2   4  9   12
***         / \
***        8  10

Maude> red inorder(
              insert(
                 insert(
                   insert(
                     insert(
                       insert(
                         insert(
                           insert(
                             insert(
                               insert(
                                 insert(
                                   insert(
                                     insert(
                                       insert(empty-tree, s(s(s(s(s(0)))))),
                                       s(s(s(s(s(s(s(0)))))))),
                                     s(0)),
                                   s(s(s(s(s(s(0))))))),
                                 s(s(s(0)))),
                               s(s(s(s(s(s(s(s(s(s(s(0)))))))))))),
                             0),
                           s(s(s(s(s(s(s(s(s(0)))))))))),
                         s(s(s(s(s(s(s(s(s(s(0))))))))))),
                       s(s(s(s(s(s(s(s(0))))))))),
                     s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))),
                   s(s(s(s(0))))),
                 s(s(0)))) .
```



```
result NeList: 0 : s(0) : s(s(0)) : s(s(s(0))) : s(s(s(s(0)))) :
    s(s(s(s(s(0))))) : s(s(s(s(s(s(0)))))) : s(s(s(s(s(s(s(0))))))) :
    s(s(s(s(s(s(s(s(0)))))))) : s(s(s(s(s(s(s(s(s(0))))))))) :
    s(s(s(s(s(s(s(s(s(s(0)))))))))) : s(s(s(s(s(s(s(s(s(s(s(0))))))))))) :
    (s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))) : []
*** 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10 : 11 : 12
```

7 Parameterization

Obviously, many of the data type specifications we have seen so far, like lists (Sections 4.3, 5.3, and 5.4), binary trees (Section 4.4), multisets (Section 5.5), or sets (Section 5.6) can be defined on data that is much more general than just natural numbers. They can indeed be defined over any data type whatsoever. Other data types like ordered lists (Section 6.3) and search trees (Section 6.4) are not so general, but can still be defined over any data type that is totally ordered.

Parameterization techniques allow us to specify more general cases for arbitrary elements satisfying some requirements that are expressed by means of theories. Views are then used to instantiate parameterized modules, to obtain, for example, a specification of lists over natural numbers from a parameterized specification of lists over an arbitrary data type.

Parameterized modules, theories and views belong to an extension of Maude called Full Maude. In the current version of the Maude system, Full Maude definitions and commands are enclosed in parentheses, and Full Maude must be loaded into Maude before they are entered (see [6]). However, the user must be aware that this may change in the future.

7.1 Theories, views, and instantiation

As we have already mentioned, parameterized datatypes use *theories* to specify the requirements that the parameter must satisfy, so that corresponding instantiations make sense. A (functional) theory is a membership equational specification whose semantics is *loose*, that is, it specifies the set of all membership algebras that satisfy the specification. In particular, equations in a theory are not used for rewriting or equational simplication and, thus, they need not satisfy any requirement about variables in the righthand side, confluence, or termination.

The simplest theory is the one requiring the existence of a sort. It is specified as follows:

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

This theory is used as requirement for the parameter of parameterized data types such as lists, multisets, sets, and binary trees.

A more complex theory is the following, requiring a total order over elements of a given sort. Notice the *new* variable `E2` in the righthand side of the first conditional equation.

```
(fth TOSET is
   protecting BOOL .
   sort Elt .
   op _<_ : Elt Elt -> Bool .
   vars E1 E2 E3 : Elt .
   eq E1 < E1 = false .
   ceq E1 < E3 = true if E1 < E2 and E2 < E3 .
   ceq E1 < E2 or E2 < E1 = true if E1 =/= E2 .
 endfth)
```

The theory `TOSET` above imports the module `BOOL` in `protecting` mode. This means that all algebras satisfying the theory `TOSET` must have a subalgebra isomorphic to the initial algebra of `BOOL`.

Theories are used in a parameterized module as in the following example:

```
(fmod LIST[X :: TRIV] is  ...  endfm)
```

where `[X :: TRIV]` denotes that `X` is the label of the formal parameter, and that it must be instantiated with modules satisfying the requirement expressed by the theory `TRIV`. The way to express this instantiation is by means of *views*. A view shows how a particular module satisfies a theory, by mapping sorts and operations in the theory to sorts and operations (or, more generally, terms) in the target module, in such a way that the induced translations on equations and membership axioms are provable in the module. In general, this requires theorem proving that is not done by the system, but in many simple cases it is completely obvious, as for example in the following view from the theory `TRIV` to the module `NAT`.

```
(view Nat from TRIV to NAT is
   sort Elt to Nat .
 endv)
```

Then, the module expression `LIST[Nat]` denotes the instantiation of the parameterized module `LIST[X :: TRIV]` by means of the above view `Nat`.

Views can also go from theories to theories, meaning an instantiation that is still parameterized. For example, we define ordered lists as a subsort of lists over a sort totally ordered; therefore we need to build lists over such sorts, that are a particular case of a sort whatsoever, but still represent a very general class that can be further instantiated, for example, to the natural numbers. The first view is


```
(view Toset from TRIV to TOSET is
   sort Elt to Elt .
 endv)
```

while the second is given by

```
(view OrdNat from TOSET to NAT is
   sort Elt to Nat .
   op _<_ to _<_ .
 endv)
```

This last view can be simplified in the sense that identity maps on operations can be omitted; however, maps must be always explicit on sorts.

Finally, note that it is possible to have more than one view from a theory to a module or to another theory; for example, on the natural numbers several different partial orders can be defined (ascending, descending, divisibility, etc.).

For much more information on parameterization and how it is implemented in the Maude system, the reader is referred to [13].

7.2 Parameterized sets

Our first parameterized module example specifies sets over any data type, which is expressed by the requirement theory `TRIV`, generalizing the version of sets over natural numbers described in Section 5.6.

In order to simplify notation, we want the usual number notation, and therefore instead of `NAT` we import the built-in module `MACHINE-INT` that provides integers and usual arithmetic operations on them.

The important point to notice is that the sorts and operations of the theory are used in the body of the parameterized module, but sorts are qualified with the name of the formal parameter; thus `Elt` becomes in this example `Elt.X`. Moreover, the new sorts introduced in the body are named in a dependent way (`Set[X]` here) so that when instantiating with a view `V`, those sorts are automatically renamed (to `Set[V]`). However, operations have exactly the same notation as in Section 5.6, and therefore equations do not change, except for number and arithmetic notation.

```
(fmod SET[X :: TRIV] is
   pr MACHINE-INT .
   sort Set[X] .
   subsorts Elt.X < Set[X] .

   op empty-set : -> Set[X] [ctor] .
   op __ : Set[X] Set[X] -> Set[X] [ctor assoc comm id: empty-set] .
```

```
    op _in_ : Elt.X Set[X] -> Bool .
    op delete : Elt.X Set[X] -> Set[X] .
    op card : Set[X] -> MachineInt .
    op _-_ : Set[X] Set[X] -> Set[X] .

    vars N N' : Elt.X .
    vars S S' : Set[X] .

    eq N N = N .
    eq N in empty-set = false .
    eq N in (N' S) = (N == N') or (N in S) .
    eq delete(N, empty-set) = empty-set .
    eq delete(N, N S) = delete(N, S) .
    ceq delete(N, N' S) = N' delete(N, S) if N =/= N' .
    eq S - empty-set = S .
    eq S - (N S') = delete(N, S) - S' .
    eq card(empty-set) = 0 .
    eq card(N S) = 1 + card(delete(N,S)) .
 endfm)
```

In the current system, the way to use an instantiation is to import it in another module, as follows, where we use a view BNat:

```
(view BNat from TRIV to BASIC-NAT is
    sort Elt to Nat .
 endv)

(fmod ANOTHER-NAT-SET is
    protecting SET[Nat] .
 endfm)
```

The internal result of this instantiation is the following module, that we only show for illustrative purposes:

```
fmod ANOTHER-NAT-SET is
  including MACHINE-INT .
  including BOOL .
  sorts Nat Set[BNat] .
  subsort Nat < Set[BNat] .
  op __ : Set[BNat] Set[BNat] -> Set[BNat]
                              [assoc comm ctor id: empty-set] .
  op empty-set : -> Set[BNat] [ctor] .
  op 0 : -> Nat [ctor].
  op _+_ : Nat Nat -> Nat .
```



```
  op _-_ : Set[BNat] Set[BNat] -> Set[BNat] .
  op s : Nat -> Nat [ctor] .
  op delete : Nat Set[BNat] -> Set[BNat] .
  op card : Set[BNat] -> MachineInt .
  op _in_ : Nat Set[BNat] -> Bool .
  vars M N N' : Nat .
  vars S S' : Set[BNat] .
  eq s(M) + N = s(M + N) .
  eq (0).Nat + N = N .
  eq S - N S' = delete(N, S) - S' .
  eq S - empty-set = S .
  eq N N = N .
  eq delete(N, N S) = delete(N, S) .
  eq delete(N, empty-set) = empty-set .
  eq card(N S) = 1 + card(delete(N, S)) .
  eq card(empty-set) = (0).MachineInt .
  eq N in N' S = N == N' or N in S .
  eq N in empty-set = false .
  ceq delete(N, N' S) = N' delete(N, S) if N =/= N' .
endfm
```

7.3 Parameterized lists and ordered lists

The following module is the parameterized version, using the theory `TRIV`, of the lists over natural numbers module in Section 4.3. The sort `Elt` in the parameter theory becomes `Elt.X`, and the new sorts `NeList[X]` and `List[X]` depend on the formal parameter `X`. All the operations and equations coincide with the ones in the more concrete module `NAT-LIST` in Section 4.3, with the exception of the operation `from_to_ : Nat Nat -> List`, which is removed, because it only makes sense to generate lists of natural numbers.

Due to parsing restrictions, some characters (`[ ] { } ,`) have to be preceded by a backquote "escape" character ` when declaring them in Full Maude.

```
(fmod LIST[X :: TRIV] is
   protecting NAT .

   sorts NeList[X] List[X] .
   subsort NeList[X] < List[X] .

   op `[`] : -> List[X] [ctor] .
   op _:_ : Elt.X List[X] -> NeList[X] [ctor] .
   op tail : NeList[X] -> List[X] .
   op head : NeList[X] -> Elt.X .
```

```
  op _++_ : List[X] List[X] -> List[X] .
  op length : List[X] -> Nat .
  op reverse : List[X] -> List[X] .
  op take_from_ : Nat List[X] -> List[X] .
  op throw_from_ : Nat List[X] -> List[X] .

  var E : Elt.X .
  var N : Nat .
  vars L L' : List[X] .

  eq tail(E : L) = L .
  eq head(E : L) = E .
  eq [] ++ L = L .
  eq (E : L) ++ L' = E : (L ++ L') .
  eq length([]) = 0 .
  eq length(E : L) = s(0) + length(L) .
  eq reverse([]) = [] .
  eq reverse(E : L) = reverse(L) ++ (E : []) .

  eq take 0 from L = [] .
  eq take N from [] = [] .
  eq take s(N) from (E : L) = E : take N from L .
  eq throw 0 from L = L .
  eq throw N from [] = [] .
  eq throw s(N) from (E : L) = throw N from L .
 endfm)
```

In the following module we instantiate `LIST[X :: TRIV]` with the view `Nat` and add the operation `from_to_` that now makes sense in the instantiated context.

```
(fmod NAT-LIST is
  protecting LIST[Nat] .
  op from_to_ : Nat Nat -> List[Nat] .
  vars N M : Nat .
  ceq from N to M = [] if M < N .
  ceq from N to M = N : from s(N) to M if not M < N .
 endfm)
```

Parameterized ordered lists need a stronger requirement, provided by the theory `TOSET` of totally ordered sets (see Section 7.1).

In the same way as the module `NAT-ORD-LIST` imported the module `NAT-LIST` in Section 6.3, it is convenient in the parameterized module for ordered lists to import the parameterized list module. However, note that we want lists for a totally ordered set, instead of lists over any set; therefore, we partially instantiate `LIST` with a view from the theory `TRIV` to the theory `TOSET`


```
(view Toset from TRIV to TOSET is
  sort Elt to Elt .
 endv)
```

and we are still left with a parameterized module and corresponding dependent sorts, with respect to the `TOSET` requirement. This is the reason justifying the notation `LIST[Toset][X]` in the `protecting` importation, as well as `NeList[Toset][X]` and `List[Toset][X]` in the names of the imported sorts. All the operations and equations coincide with the ones in the more concrete module `NAT-ORD-LIST` in Section 6.3.

```
(fmod ORD-LIST[X :: TOSET] is
  protecting LIST[Toset][X] .

  sorts OrdList[X] NeOrdList[X] .
  subsorts NeOrdList[X] < OrdList[X] NeList[Toset][X] < List[Toset][X] .

  op insertion-sort : List[Toset][X] -> OrdList[X] .
  op insert-list : OrdList[X] Elt.X -> OrdList[X] .

  op mergesort : List[Toset][X] -> OrdList[X] .
  op merge : OrdList[X] OrdList[X] -> OrdList[X] .

  op quicksort : List[Toset][X] -> OrdList[X] .
  op leq-elems : List[Toset][X] Elt.X -> List[Toset][X] .
  op gr-elems : List[Toset][X] Elt.X -> List[Toset][X] .

  vars E E' : Elt.X .
  vars L L' : List[Toset][X] .
  vars OL OL' : OrdList[X] .
  var NEOL : NeOrdList[X] .
  var NEL : NeList[Toset][X] .

  mb [] : OrdList[X] .
  mb (E : []) : NeOrdList[X] .
  cmb (E : NEOL) : NeOrdList[X] if not head(NEOL) < E .

  eq insert-list([], E') = E' : [] .
  ceq insert-list(E : OL, E') = E' : E : OL if E' < E or E' == E .
  ceq insert-list(E : OL, E') = E : insert-list(OL, E') if E < E' .

  eq insertion-sort([]) = [] .
  eq insertion-sort(E : L) = insert-list(insertion-sort(L), E) .
```

```
  eq mergesort([]) = [] .
  eq mergesort(E : []) = E : [] .
  ceq mergesort(L)
    = merge(mergesort(take (length(L) div s(s(0))) from L),
            mergesort(throw (length(L) div s(s(0))) from L))
     if s(0) < length(L) .

  eq merge(OL, []) = OL .
  eq merge([], OL) = OL .
  ceq merge(E : OL, E' : OL') = E : merge(OL, E' : OL') if not E' < E .
  ceq merge(E : OL, E' : OL') = E' : merge(E : OL, OL') if E' < E .

  eq quicksort([]) = [] .
  eq quicksort(E : L)
   = quicksort(leq-elems(L, E)) ++ (E : quicksort(gr-elems(L, E))) .

  eq leq-elems([], E') = [] .
  ceq leq-elems(E : L, E') = E : leq-elems(L, E') if not (E' < E) .
  ceq leq-elems(E : L, E') = leq-elems(L, E') if E' < E .
  eq gr-elems([], E') = [] .
  ceq gr-elems(E : L, E') = gr-elems(L, E') if not (E' < E) .
  ceq gr-elems(E : L, E') = E : gr-elems(L, E') if E' < E .
 endfm)
```

Finally, to instantiate the parameterized module with respect to the theory TOSET we need a view from it to a module:

```
(view OrdNat from TOSET to NAT is
  sort Elt to Nat .
  op _<_ to _<_ .
 endv)

(fmod NAT-ORD-LIST is
  pr ORD-LIST[OrdNat] .
  op from_to_ : Nat Nat -> OrdList[OrdNat] .
  vars N M : Nat .
  ceq from N to M = [] if M < N .
  ceq from N to M = N : from s(N) to M if not M < N .
 endfm)
```

7.4 Parameterized binary trees and search trees

The parameterization for binary trees follows exactly the same pattern as the one for lists: the requirement theory is TRIV, sorts are renamed, and operations and

equations are the same as for the particular case defined in Section 4.4.

```
(fmod BIN-TREE[X :: TRIV] is
  protecting LIST[X] .

  sorts NeBinTree[X] BinTree[X] .
  subsort NeBinTree[X] < BinTree[X] .

  op empty : -> BinTree[X] [ctor] .
  op _`[_`]_ : BinTree[X] Elt.X BinTree[X] -> NeBinTree[X] [ctor] .
  ops left right : NeBinTree[X] -> BinTree[X] .
  op root : NeBinTree[X] -> Elt.X .
  op depth : BinTree[X] -> Nat .
  ops leaves preorder inorder postorder : BinTree[X] -> List[X] .

  var E : Elt.X .
  vars L R : BinTree[X] .
  vars NEL NER : NeBinTree[X] .

  eq left(L [E] R) = L .
  eq right(L [E] R) = R .
  eq root(L [E] R) = E .
  eq depth(empty) = 0 .
  eq depth(L [E] R) = s(0) + max(depth(L), depth(R)) .

  eq leaves(empty) = [] .
  eq leaves(empty [E] empty) = E : [] .
  eq leaves(NEL [E] R) = leaves(NEL) ++ leaves(R) .
  eq leaves(L [E] NER) = leaves(L) ++ leaves(NER) .

  eq preorder(empty) = [] .
  eq preorder(L [E] R) = E : (preorder(L) ++ preorder(R)) .
  eq inorder(empty) = [] .
  eq inorder(L [E] R) = inorder(L) ++ (E : inorder(R)) .
  eq postorder(empty) = [] .
  eq postorder(L [E] R) = postorder(L) ++ (postorder(R) ++ (E : [])) .
 endfm)
```

Although in the parameterization of binary search trees we could follow the same pattern of ordered lists, and generalize the search trees in Section 6.4 with respect to a total order, we define here a slightly different version of search trees (thinking of them as dictionaries or association tables), with pairs <key;contents> in the nodes, mainly to make the parameterization much more interesting. The search tree structure is with respect to a total order on keys, but contents can be over an arbitrary sort.

In the previous concrete version of search trees, when inserting an element already in the tree, the tree was not modified (see Section 6.4). Now, we insert a pair $\langle k, c\rangle$, and when the key k already appears in the tree in a pair $\langle k, c'\rangle$, insertion takes place by combining the contents c' and c. This combination can be addition of multiplicities, replacing the first with the second, just forgetting the second, etc. Therefore, as part of the requirement theory we have a `combine` operation on the sort `Contents`, which will be instantiated in some way or another depending on the particular case.

Moreover, in addition to operations for insertion, search, and deletion as before, we have a `lookup` operation that returns the contents associated to a given key, when the key appears in the tree; if it does not appear, a special value `not-found` is returned. The theory also takes care of requiring this special value in a supersort `Contents?` of the sort `Contents`.

Note that the theory `PAIR` imports a renamed copy of the theory `TOSET`, where the sort `Elt` is renamed to `Key` (the syntax for renamings uses the `*` symbol and is based on maps for sorts and operations, just like the syntax for views).

```
(fth PAIR is
   including TOSET * (sort Elt to Key) .
   sorts Pair Contents Contents? .
   subsort Contents < Contents? .
   op <_;_> : Key Contents -> Pair .
   op key : Pair -> Key .
   op contents : Pair -> Contents .
   op combine : Contents Contents -> Contents .
   op not-found : -> Contents? .
   var K : Key .
   var C : Contents .
   eq key(< K ; C >) = K .
   eq contents(< K ; C >) = C .
 endfth)

(view Pair from TRIV to PAIR is
   sort Elt to Pair .
 endv)
```

We also need a partial instantiation of binary trees by means of the above view `Pair` between the two relevant theories.

```
(fmod SEARCH-TREE[X :: PAIR] is
   protecting BIN-TREE[Pair][X] .
```



```
sorts SearchTree[X] NeSearchTree[X] .
subsorts NeSearchTree[X] < SearchTree[X] < BinTree[Pair][X] .
subsort NeSearchTree[X] < NeBinTree[Pair][X] .

op insert : SearchTree[X] Pair.X -> SearchTree[X] .
op lookup : SearchTree[X] Key.X -> Contents?.X .
op delete : SearchTree[X] Key.X -> SearchTree[X] .
op find : SearchTree[X] Key.X -> Bool .
ops min max : NeSearchTree[X] -> Pair.X .

vars E E' : Pair.X .
vars L R : SearchTree[X] .
vars L' R' : NeSearchTree[X] .
var K : Key.X .
vars C C' : Contents.X .

mb empty : SearchTree[X] .
mb empty [E] empty : NeSearchTree[X] .
cmb L' [E] empty : NeSearchTree[X] if key(max(L')) < key(E) .
cmb empty [E] R' : NeSearchTree[X] if key(E) < key(min(R')) .
cmb L' [E] R' : NeSearchTree[X]
    if key(max(L')) < key(E) and key(E) < key(min(R')) .

eq insert(empty, E) = empty [E] empty .
eq insert(L [< K ; C >] R, < K ; C' >) = L [< K ; combine(C, C') >] R .
ceq insert(L [E] R, E') = insert(L, E') [E] R if key(E') < key(E) .
ceq insert(L [E] R, E') = L [E] insert(R, E') if key(E) < key(E') .

eq lookup(empty, K) = not-found .
eq lookup(L [< K ; C >] R, K) = C .
ceq lookup(L [E] R, K) = lookup(L, K) if K < key(E) .
ceq lookup(L [E] R, K) = lookup(R, K) if key(E) < K .

eq delete(empty, K) = empty .
ceq delete(L [E] R, K) = delete(L, K) [E] R if K < key(E) .
ceq delete(L [E] R, K) = L [E] delete(R, K) if key(E) < K .
eq delete(empty [< K ; C >] R, K) = R .
eq delete(L [< K ; C >] empty, K) = L .
eq delete(L' [< K ; C >] R', K) =
     L' [min(R')] delete(R', key(min(R'))) .

eq find(empty, K) = false .
eq find(L [< K ; C >] R, K) = true .
```

```
  ceq find(L [E] R, K) = find(L, K) if K < key(E) .
  ceq find(L [E] R, K) = find(R, K) if key(E) < K .

  eq min(empty [E] R) = E .
  eq min(L' [E] R) = min(L') .
  eq max(L [E] empty) = E .
  eq max(L [E] R') = max(R') .
 endfm)
```

We need some auxiliary, but generally useful, modules before instantiating the previous module.

First, we consider a parameterized module `DEFAULT[X :: TRIV]` that simply adds a supersort to the sort `Elt` of `TRIV` and a "default" constant `null`.

```
(fmod DEFAULT[X :: TRIV] is
  sort Default[X] .
  subsort Elt.X < Default[X] .
  op null : -> Default[X] .
 endfm)

(view Qid from TRIV to QID is
  sort Elt to Qid .
 endv)

(view Default`[Qid`] from TRIV to DEFAULT[Qid] is
  sort Elt to Default[Qid] .
 endv)
```

A very useful *module constructor* is the operation `TUPLE(`n`)`: Given a natural number $n \geq 2$, `TUPLE(`n`)` is a parameterized module over several copies of the theory `TRIV` that specifies tuples of size n with a tuple constructor and the n corresponding projections.

We instantiate the theory `PAIR` to pairs of natural numbers and (quoted) identifiers, where we have previously added the default value by using the `DEFAULT` module above, and where the combination operation becomes concatenation `conc` of identifiers.

```
(view Tuple`[Nat`,Default`[Qid`]`] from PAIR
                                   to TUPLE(2)[Nat, Default`[Qid`]] is
  sort Pair to Tuple[Nat, Default`[Qid`]] .
  sort Key to Nat .
  sort Contents to Qid .
  sort Contents? to Default[Qid] .
  op key to p1_ .
```



```
  op contents to p2_ .
  op <_;_> to `(_`,_`) .
  op combine to conc .
  op not-found to null .
 endv)

(fmod ANOTHER-NAT-SEARCH-TREE is
  protecting SEARCH-TREE[Tuple`[Nat`,Default`[Qid`]`]] .
 endfm)

***       5f
***    /      \
***   1b       7h
***  /  \      / \
*** 0a   3d   6g  11l
***     /   \    /   \
***    2c   4e  9j   12m
***            / \
***          8i   10k

Maude>
(red inorder(
        insert(
          insert(
            insert(
              insert(
                insert(
                  insert(
                    insert(
                      insert(
                        insert(
                          insert(
                            insert(
                              insert(
                                insert(empty, (s(s(s(s(s(0))))), 'f)),
                                (s(s(s(s(s(s(s(0))))))), 'h)),
                              (s(0), 'b)),
                            (s(s(s(s(s(s(0)))))), 'g)),
                          (s(s(s(0))), 'd)),
                       (s(s(s(s(s(s(s(s(s(s(s(0))))))))))), 'l)),
                      (0, 'a)),
                    (s(s(s(s(s(s(s(s(s(0))))))))), 'j)),
                 (s(s(s(s(s(s(s(s(s(s(0)))))))))), 'k)),
                (s(s(s(s(s(s(s(s(0)))))))), 'i)),
```

```
           (s(s(s(s(s(s(s(s(s(s(s(0))))))))))), 'm)),
          (s(s(s(s(0)))), 'e)),
         (s(s(0)), 'c))) .)

result NeList`[Pair`]`[Tuple`[Nat`,Default`[Qid`]`]`] :
    (0 , 'a) : (s(0) , 'b) : (s(s(0)) , 'c) : (s(s(s(0))) , 'd) :
    (s(s(s(s(0)))) , 'e) : (s(s(s(s(s(0))))) , 'f) :
    (s(s(s(s(s(s(0)))))) , 'g) : (s(s(s(s(s(s(s(0))))))) , 'h) :
    (s(s(s(s(s(s(s(s(0)))))))) , 'i) : (s(s(s(s(s(s(s(s(s(0))))))))) , 'j) :
    (s(s(s(s(s(s(s(s(s(s(0)))))))))) , 'k) :
    (s(s(s(s(s(s(s(s(s(s(s(0))))))))))) , 'l) :
    (s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))) , 'm) : [ ]
```

7.5 General trees

The following example is simpler from the parameterization point of view, although more complex from the data structure point of view. We specify *general trees* which store in their nodes data elements in an arbitrary sort, specified using the theory TRIV. Since the number of children for each node is variable, we have a list structure for the children. However, children are themselves general trees and, therefore, there is a recursive dependency that does not allow us to import a parameterized list module instantiated over the trees being defined. Instead, we have to specify simultaneously both data types, trees and lists of trees that we call *forests*.

Some operations are the same as the operations on lists, and there is only one constructor `_[_]` for trees with corresponding selectors `root` and `children`. Note that there is no empty tree in this case, although there is an empty forest, that is, an empty list of children.

The operations `depth` and `degree` calculate the depth of a tree and the maximum number of children over all nodes in the tree. `preorder` and `postorder` are standard traversals over general trees. Each one of these operations is defined by means of an auxiliary operation on forests, defined by structural induction, and mutually recursive with respect to the main operation (see the equations for `depth` and `depth-forest`, for example).

```
(fmod TREE[X :: TRIV] is
   protecting LIST[X] .

   sorts Tree[X] Forest[X] .

   op _`[_`] : Elt.X Forest[X] -> Tree[X] [ctor] .
   op empty-forest : -> Forest[X] [ctor] .
   op _:_ : Tree[X] Forest[X] -> Forest[X] [ctor] .
```



```
   op root : Tree[X] -> Elt.X .
   op children : Tree[X] -> Forest[X] .
   ops depth degree : Tree[X] -> Nat .
   ops depth-forest degree-forest : Forest[X] -> Nat .
   op #children : Tree[X] -> Nat .
   op length : Forest[X] -> Nat .
   op leaf? : Tree[X] -> Bool .

   ops preorder postorder : Tree[X] -> List[X] .
   ops preorder-forest postorder-forest : Forest[X] -> List[X] .

   var E : Elt.X .
   var T : Tree[X] .
   var F : Forest[X] .

   eq root(E [F]) = E .
   eq children(E [F]) = F .
   eq depth(E [F]) = s(0) + depth-forest(F) .
   eq depth-forest(empty-forest) = 0 .
   eq depth-forest(T : F) = max(depth(T), depth-forest(F)) .
   eq length(empty-forest) = 0 .
   eq length(T : F) = s(0) + length(F) .
   eq #children(E [F]) = length(F) .
   eq leaf?(T) = #children(T) == 0 .
   eq degree(E [F]) = max(length(F), degree-forest(F)) .
   eq degree-forest(empty-forest) = 0 .
   eq degree-forest(T : F) = max(degree(T), degree-forest(F)) .

   eq preorder(E [F]) = E : preorder-forest(F) .
   eq preorder-forest(empty-forest) = [] .
   eq preorder-forest(T : F) = preorder(T) ++ preorder-forest(F) .
   eq postorder(E [F]) = postorder-forest(F) ++ (E : []) .
   eq postorder-forest(empty-forest) = [] .
   eq postorder-forest(T : F) = postorder(T) ++ postorder-forest(F) .
 endfm)
```

7.6 Parameterized paths

This example is a parameterized version of the module defining paths over a graph in Section 6.2. The requirement theory GRAPH specifies an arbitrary graph, that is, two sorts, for nodes and edges, and source and target operations. The module A-GRAPH in Section 6.2 obviously satisfies this theory. Given an arbitrary graph, it makes sense to define the paths over the graph. This is done in the parameterized module

PATH[X :: GRAPH], where, as in previous examples, operations and equations do not change with respect to the unparameterized version, while sorts are renamed according to whether they come from the theory or are introduced as new sorts in the body.

```
(fth GRAPH is
  sorts Edge Node .
  ops source target : Edge -> Node .
endfth)

(fmod PATH[X :: GRAPH] is
  protecting NAT .

  sorts Path[X] Path?[X] .
  subsorts Edge.X < Path[X] < Path?[X] .
  op _;_ : Path?[X] Path?[X] -> Path?[X] [ctor assoc] .
  ops source target : Path[X] -> Node.X .
  op length : Path[X] -> Nat .

  var E : Edge.X .
  var P : Path[X] .

  cmb (E ; P) : Path if target(E) == source(P) .

  eq source(E ; P) = source(E) .
  eq target(P ; E) = target(E) .
  eq length(E) = s(0) .
  eq length(E ; P) = s(0) + length(P) .
endfm)

(view AGraph from GRAPH to A-GRAPH is
   sort Edge to Edge .
   sort Node to Node .
   op source to source .
   op target to target .
 endv)

(fmod A-GRAPH-PATH is
   protecting PATH[AGraph] .
 endfm)

Maude> (red target(a ; b ; c) .)
result Error ( Node ) : target ( a ; b ; c )
```


8 Rewriting logic specifications

Equational logic, in all its variants, is a logic to reason about *static* data types, where the notions of “before” and “after” do not make sense since, due to the symmetry of equality, all change is reversible. By removing the symmetry rule of deduction in equational logic, equations are no longer symmetric and they become oriented, as in equational rewriting. We can take an important step further by dropping symmetry and the equational interpretation of rules, and interpreting a rule $t \to t'$ *computationally* as a local concurrent transition of a system, and *logically* as an inference step from formulas of type t to formulas of type t'. In this way, we arrive at the main idea behind *rewriting logic*.

Rewriting logic is a logic of *becoming* or *change*, that allows us to specify the dynamic aspects of systems in a very general sense. Moreover, by allowing rewriting over congruence classes modulo some structural axioms, we can understand a “term” $[t]$ as a *proposition* or *formula* that asserts being in a certain *state* having a certain *structure*. For example, for Petri nets (see Section 8.3) the corresponding structure is that of multisets, usually called markings in that context.

8.1 Rewriting logic

A rewriting logic *signature* is an equational specification, so that rewriting logic is parameterized by the choice of its underlying equational logic. For Maude the underlying equational logic is membership equational logic, our signatures will be of the form (Ω, E), where $\Omega = (K, \Sigma, S)$ is a membership equational logic signature and E is a set of (conditional) membership axioms and equations. Such a signature (Ω, E) makes explicit the set of equations in order to emphasize that rewriting will operate on congruence classes of terms *modulo* E. This is precisely what Maude does, using the equational attributes given in operation declarations (associativity, commutativity, and identity) to rewrite modulo such axioms; see Section 5.8.

Given a signature (Ω, E), sentences of the logic are sequents (called *rewrites*) of the form $[t]_E \longrightarrow [t']_E$, where t and t' are terms possibly involving some variables.

A *rewriting logic specification* $\mathcal{R}$ is a tuple $\mathcal{R} = (\Omega, E, L, R)$ where (Ω, E) is a signature, L is a set of labels, and R is a set of labelled *rewrite rules* written

$$r : [t]_E \longrightarrow [t']_E$$

where r is a label and $[t]_E$ and $[t']_E$ are congruence classes of terms in $\mathcal{T}_{\Omega,E}(X)$, with $X = \{x_1, \ldots, x_n, \ldots\}$ a countably infinite set of variables.

The logic and its theoretical results can be extended to more expressive conditional rules of the form

$$r : [t] \longrightarrow [t'] \;\; \mathit{if} \;\; [u_1] \longrightarrow [v_1] \;\wedge \ldots \wedge\; [u_k] \longrightarrow [v_k],$$

but in the current version of Maude only Boolean conditions are supported for rules of the form $r : [t] \longrightarrow [t']$ *if* b, where b is a Boolean term. A future Maude release will support general rules with rewrites, equations, and memberships in their conditions.

There are logical rules of deduction for rewriting logic that can be obtained from the logical rules for (membership) equational logic by dropping the symmetry rule, as we have already mentioned (for an unsorted version of the logic, its model-theoretic semantics, and soundness, completeness, and initiality results, see [21]). Since (congruence classes of) terms describe states of a system, the rewrite rules in a specification describe which *elementary local transitions* are possible in the distributed state by concurrent local transformations. The deduction rules of rewriting logic allow us to reason correctly about which *general* concurrent transitions are possible in a system satisfying such a description. Thus, computationally, each rewriting step is a parallel local transition in a concurrent system. Alternatively, however, we can adopt a logical viewpoint instead, and regard the deduction rules of rewriting logic as *metarules* for correct deduction in a *logical system*. Logically, each rewriting step is a logical *entailment* in a formal system. The computational and the logical viewpoints under which rewriting logic can be interpreted can be summarized as follows:

State	$\leftrightarrow$	*Term*	$\leftrightarrow$	*Proposition*
Transition	$\leftrightarrow$	*Rewriting*	$\leftrightarrow$	*Deduction*
Distributed Structure	$\leftrightarrow$	*Algebraic Structure*	$\leftrightarrow$	*Propositional Structure*

Rewriting logic has also an interesting model theory, generalizing equational algebras to systems with states and transitions corresponding to terms and rewrite steps. This model theory enjoys properties of soundness and completeness with respect to logical deduction, and of existence of initial models. Moreover, by regarding an equational specification as a rewrite theory whose set of rules is empty, rewriting logic is a conservative extension of (membership) equational logic [20].

System modules in Maude correspond to rewrite theories in rewriting logic. In the unparameterized case a system module's semantics is the corresponding initial model. From a systems perspective this model describes all the concurrent behaviors that the system so axiomatized can exhibit.

A rewrite theory has both rules and equations, so that rewriting is performed *modulo* such equations. However, this does not mean that the Maude implementation must have a matching algorithm for each equational theory that a user might specify, which is impossible, since matching modulo an arbitrary theory is undecidable. What we instead require for rewrite theories in system modules is that:

- The equations are divided into a set A of structural axioms, for which matching algorithms exist in the Maude implementation and a set E of equations that are Church-Rosser and terminating *modulo* A; that is, the equational part must be equivalent to a functional module.

- The rules R in the module are *coherent* [30] with the equations E modulo A. This means that appropriate critical pairs exist between rules and equations, allowing us to intermix rewriting with rules and rewriting with equations without loosing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken. In this way, we get the effect of rewriting modulo $E \cup A$ with just a matching algorithm for A. In particular, a simple strategy available in these circumstances is to always reduce to canonical form using E before applying any rule in R. This is precisely the strategy adopted by the Maude interpreter.

One of the key goals of rewriting logic from its beginning has been to provide a *semantic* and *logical framework* in which many different logics, models of computation, and languages can be represented, can be given a precise semantics, and can be executed. There is by now very extensive evidence supporting the claim that rewriting logic is indeed a very flexible and simple logical and semantic framework, and some of the examples below are simple representations of these ideas.

For more details on rewriting logic and its applications, we refer the reader to the papers [21, 22, 20, 23, 25, 9]

8.2 Transition systems

As a first example of the idea of rewriting logic as a semantic framework, we show how to specify in Maude transition systems, a heavily used model of computation. Consider the graph of Figure 1 in Section 6.2 as a transition system: nodes become states and edges transitions between such states.

The following system module specifies such a transition system as a rewrite theory. There are only constant operations corresponding to the states, with no other operations, no variables, and no equations. Each transition becomes a rewrite rule, and the transition name becomes the corresponding rule label.

```
mod A-TRANSITION-SYSTEM is
  sort State .
  ops n1 n2 n3 n4 n5 : -> State [ctor] .

  rl [a] : n1 => n2 .
  rl [b] : n1 => n3 .
  rl [c] : n3 => n4 .
  rl [d] : n4 => n2 .
  rl [e] : n2 => n5 .
  rl [f] : n2 => n1 .
```

```
endm
```

Note that this specification is not confluent since there are, for example, two transitions out of `n2` that are not joinable, and it is not terminating either, since there are cycles (see Figure 1) creating infinite computations. Therefore, as opposed to equational simplification for functional modules, rewriting can go in many undesired directions. We will see in Section 11 how rewriting can be controlled by means of *strategies* defined by the user. However, the Maude interpreter provides a *default strategy* for executing expressions in system modules, provided by the `rewrite` command, or, in abbreviated form, `rew`. Due to the possibility of nontermination, this command admits as argument a bound on the number of rule applications; for example,

```
Maude> rew [10] n3 .
result State: n5
```

8.3 Petri nets

Petri nets constitute another model of concurrent computation, used both at the theoretical and practical levels [27]. We illustrate again by means of one example how place/transition Petri nets can be specified as system modules in Maude.

Let us consider a Petri net modelling a small library, where a token represents a book, that can be in several different states: just bought (j), available (a), borrowed (b), requested (r), and not available (n). The possible transitions are the following:

- *buy*: When there are four accumulated requests, the library places an order to buy two copies of each requested book (although this representation does not distinguish among different books or copies of the same book).
- *file*: A book just bought is filed, making it available.
- *borr*: An available book can be borrowed.
- *ret*: A borrowed book can be returned.
- *lose*: A borrowed book can become lost, and thus no longer available.
- *disc*: An available book is discarded because of its bad condition, and thus it is no longer available either.
- *req1*: A user may place a request to buy a non available book, but only when there are two accumulated requests these are honored.
- *req2*: The library may decide to buy a new book, thus creating a new token in the system, with no precondition in this representation.

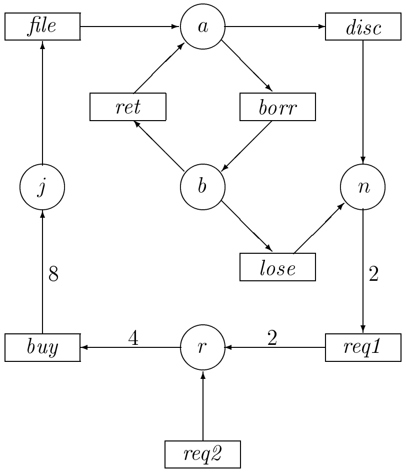


Figure 2: A Petri net model of a library.

The corresponding graphical representation for this Petri net in the usual style is depicted in Figure 2.

A marking on a Petri net is a multiset over its set of places, denoting the available resources (or tokens) in each place. A transition goes from the marking representing the resources it consumes (its preset) to the marking representing the resources it produces (its postset). Therefore, in the following system module, first we define a multiset structure for markings by means of the corresponding structural axioms (note that multiset union has empty syntax, as in Section 5.5). Then, each transition gives rise to a rewrite rule from its preset to its poset, in the same way as we did for transition systems in the previous section. The only exception to this general rule is the rule corresponding to the transition *req2*; instead of a rule of the form `1 => r`, we have to write `M => M r` with `M` a variable of sort `Marking`. The reason is that the constant `1` is not a `__`-subterm of any marking except itself, according to the definition in Section 5.8, and thus it would be imposible to apply the rule `1 => r` with extension (see Section 5.8).

```
mod LIBRARY-PETRI-NET is
  sorts Place Marking .
  subsort Place < Marking .
  op 1 : -> Marking [ctor] .
  op __ : Marking Marking -> Marking [ctor assoc comm id: 1] .
```

```
  ops a b n r j : -> Place [ctor] .

  var M : Marking .

  rl [buy] : r r r r => j j j j j j j j .
  rl [file] : j => a .
  rl [borr] : a => b .
  rl [ret] : b => a .
  rl [lose] : b => n .
  rl [disc] : a => n .
  rl [req1] : n n => r r .
  rl [req2] : M => M r .
endm
```

It can easily be seen that there is a concurrent computation in the net from a marking M to a marking M' if and only if there is a rewriting from M to M' using the rules in the Petri net as a system module.

Again, note that computations may be nonterminating, for example, because a book can be forever in the borrow-return cycle. This particular net happens to be confluent by chance, because in the two places where there is nondeterminism (an available book can be either borrowed or discarded, and a borrowed book can be either returned or lost), it is possible to follow another path to the same place; but it is obvious that in an arbitrary net confluence is not a required or even desirable property, precisely because of the interest in modelling nondeterminism.

The following rewriting starts in the empty marking, and finishes in the marking consisting of 16 books just bought.

```
Maude> rew [10] 1 .
result Marking: j j j j j j j j j j j j j j j j
```

8.4 Blocks world

A generalization of Petri net computations is provided by conjunctive planning problems, where the states are described by means of some kind of conjunction of propositions describing basic facts. A typical example in artificial intelligence circles is the blocks world. In the present version there is a table on top of which we have the blocks, which can be moved only by means of a robot arm. We have as basic propositions some predicates whose intuitive meaning is given in the accompanying comments.

The rule `pickup` describes how the robot arm, being empty, takes a block from the table, while `putdown` corresponds to the inverse move. The pair of rules `unstack` and `stack` describe similar moves when the block is on top of another one. As

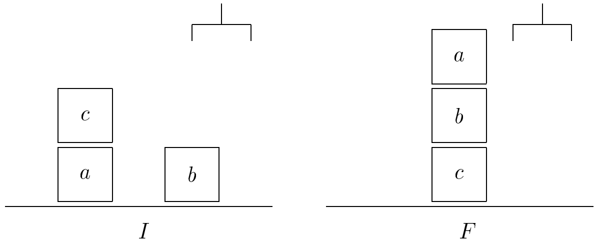


Figure 3: Initial and final states in a world with three blocks.

opposed to the Petri net transitions, note that here the states in rules have variables, so that they can be instantiated with identifiers for blocks.

```
mod BLOCKS-WORLD is
  protecting QID .
  sorts BlockId Prop State .
  subsort Qid < BlockId .
  subsort Prop < State .

  op table : BlockId -> Prop [ctor] .      *** block is on the table
  op on : BlockId BlockId -> Prop [ctor] . *** block A is on block B
  op clear : BlockId -> Prop [ctor] .      *** block is clear
  op hold : BlockId -> Prop [ctor] .       *** robot arm holds the block
  op empty : -> Prop [ctor] .              *** robot arm is empty
  op 1 : -> State [ctor] .
  op _&_ : State State -> State [ctor assoc comm id: 1] .

  vars X Y : BlockId .

  rl [pickup] : empty & clear(X) & table(X) => hold(X) .
  rl [putdown] : hold(X) => empty & clear(X) & table(X) .
  rl [unstack] : empty & clear(X) & on(X,Y) => hold(X) & clear(Y) .
  rl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X,Y) .
endm
```

Consider for example the states described in Figure 3. The initial state I on the left and the final state F on the right are respectively described by the following two terms of sort `State`:

```
  empty & clear('c) & clear('b) & table('a) & table('b) & on('c,'a)

  empty & clear('a) & table('c) & on('a,'b) & on('b,'c)
```

The fact that the "sequential plan" (in a self-explanatory intuitive notation)

```
unstack(c,a); putdown(c); pickup(b); stack(b,c); pickup(a); stack(a,b)
```

moves the blocks from state I to state F corresponds directly to a sequence of computational rewrite steps applying the corresponding rewrite rules.

In Section 11 we will use a simpler version of the blocks world with no robot arm, in order to illustrate user-defined strategies to control rewriting, either to avoid nontermination or to choose a specific "plan" for a computational task or logical proof. A similar simpler version is used in Section 9.5 to illustrate an object-oriented approach.

8.5 Sequent calculus for propositional logic

The following example is a small illustration of the use of rewriting logic as a logical framework, describing the representation within rewriting logic of a sequent calculus for classical propositional logic, where the basic atoms are denoted by (quoted) identifiers.

We can take advantage of several properties enjoyed by classical logic in order to have a more succint and elegant sequent calculus. First, using the double negation and De Morgan laws, we can push negation to the atomic level. Also, we can use one-sided sequents of the form $\vdash A_1, \ldots, A_n$, instead of the more usual double-sided sequents of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, because such a sequent is equivalent to the one-sided sequent $\vdash \sim A_1, \ldots, \sim A_n, B_1, \ldots, B_m$, where the symbol $\sim$ denotes negation. One of the advantages of this presentation is that the number of rules is reduced to a half. Furthermore, the contraction and permutation structural rules can be built into the sequent data structure by using (nonempty) sets of propositions instead of lists.

The first *functional* module defines the syntax of classical propositional logic, using the connectives of conjunction, disjunction, and negation. Equations are used in order to push negation to the atomic level. A one-sided sequent is defined as a set of propositions of the form `|- A1,...,An`.

```
fmod PROP-LOG is
  protecting QID .
  sorts Atom Prop .
  subsorts Qid < Atom < Prop .          *** atomic propositions
  ops tt ff : -> Prop [ctor] .          *** truth values
  op ~_ : Prop -> Prop [ctor] .         *** negation
  op _\/_ : Prop Prop -> Prop [ctor] .  *** disjunction
  op _/\_ : Prop Prop -> Prop [ctor] .  *** conjunction
```



```
  vars P Q : Prop .
  eq ~ tt = ff .
  eq ~ ff = tt .
  eq ~ ~ P = P .                        *** double negation
  eq ~ (P \/ Q) = ~ P /\ ~ Q .          *** De Morgan laws
  eq ~ (P /\ Q) = ~ P \/ ~ Q .

  sorts PropSet Sequent .
  subsort Prop < PropSet .
  op _,_ : PropSet PropSet -> PropSet [ctor assoc comm] .
  op |-_ : PropSet -> Sequent [ctor] .
  var S : PropSet .
  eq S,S = S .
endfm
```

The idempotency equation `eq S,S = S` is unproblematic in the module above because the set union operation `_,_` does not have an identity attribute.

Having defined all the necessary syntax, the following *system module* defines the sort `Configuration` of multisets of sequents representing the premises and conclusion of deduction rules, and then gives the corresponding one-sided sequent calculus rules for classical propositional logic as rewrite rules on the sort `Configuration`. Using the fact that text beginning with `---` is a comment in Maude, rules are displayed in such a way as to emphasize the correspondence with the usual inference rule presentation in logic textbooks.

```
mod SEQUENT-RULES-PROP-LOG is
  protecting PROP-LOG .
  sort Configuration .
  subsort Sequent < Configuration .
  op empty : -> Configuration [ctor] .
  op __ : Configuration Configuration -> Configuration
                                        [ctor assoc comm id: empty] .
  vars R S : PropSet .
  vars P Q : Prop .

  rl [Identity] :          empty
                    => -------------
                         |- (P, ~ P) .

  rl [Cut] :             |- (R, P) |- (S, ~ P)
                    => ---------------------
                              |- (R, S) .

  rl [Weakening] :         |- R
```

```
                       => -----------
                          |- (R, P) .

  rl [Disjunction] :         |- (R, P, Q)
                       => ------------------
                          |- (R, (P \/ Q)) .

  rl [Conjunction] :       |- (R, P) |- (S, Q)
                       => ---------------------
                          |- (R, S, (P /\ Q)) .

  rl [Truth] :              empty
                       => -------
                          |- tt .
endm
```

This general method of viewing sequents as rewrite rules can be applied to systems more general than traditional sequent calculi. Thus, a "sequent" can, for example, be a sequent presentation of natural deduction, a term assignment system, or even any predicate defined by structural induction in some way such that the proof is a kind of tree, as done, for example, in structural operational semantics [17], including type-checking systems. The general idea is to map a rule in the "sequent" system to a rewrite rule over a "configuration" of sequents or predicates, in such a way that the rewriting relation corresponds to provability of such a predicate [20].

The user must be aware that this representation is good from the mathematical point of view but it is not the most suited for doing proof search. There are general techniques using strategies that can be applied for this [29].

8.6 Lambda calculus

This section shows that higher-order (in the sense of lambda calculi) computational models can also be represented inside rewriting logic. We want to represent the reduction over (untyped) λ-calculus terms as rewriting inside rewriting logic. The simple idea is to specify the syntax as a functional module as we have seen before, and then in a system module add the usual β and η reduction rules. The problem is that β has in its righthand side a substitution operation that is usually defined at the mathematical metalevel, outside the syntax of the calculus. To solve this, we follow the approach of so called *explicit* substitution calculi, in which substitution is just another term constructor defined by means of the usual equations.

Our presentation makes an interesting use of parameterization. The calculus is defined with respect to an arbitrary set of variables; the only requirement is the

possibility of always obtaining *new* variables, which are used in renaming bound variables to avoid the capture of free variables. The operation that calculates the free variables of a term is also in the representation, inside the calculus, instead of at the mathematical metalevel.

```
(fth VAR is
   protecting BOOL .
   sorts Var VarSet .
   subsort Var < VarSet .               *** singleton sets
   op empty-set : -> VarSet .           *** empty set
   op _U_ : VarSet VarSet -> VarSet [assoc comm id: empty-set] .
                                        *** set union
   op _in_ : Var VarSet -> Bool .       *** membership test
   op _-_ : VarSet VarSet -> VarSet .   *** set difference
   op new : VarSet -> Var .             *** new variable

   vars E E' : Var .
   vars S S' : VarSet .

   eq E U E = E .
   eq E in empty-set = false .
   eq E in E' U S = (E == E') or (E in S) .
   eq empty-set - S = empty-set .
   eq (E U S) - S' = if E in S' then S - S' else E U (S - S') fi .
   eq new(S) in S = false .
 endfth)

(fmod LAMBDA[X :: VAR] is
   sort Lambda[X] .
   subsort Var.X < Lambda[X] .             *** variables
   op \_._ : Var.X Lambda[X] -> Lambda[X] [ctor] .
                                           *** lambda abstraction
   op __ : Lambda[X] Lambda[X] -> Lambda[X] [ctor] .
                                           *** application
   op _`[_/_`] : Lambda[X] Lambda[X] Var.X -> Lambda[X] .
                                           *** substitution
   op fv : Lambda[X] -> VarSet.X .         *** free variables

   vars X Y : Var.X .
   vars M N P : Lambda[X] .

   *** Free variables
   eq fv(X) = X .
   eq fv(\ X . M) = fv(M) - X .
```

```
  eq fv(M N) = fv(M) U fv(N) .
  eq fv(M [N / X]) = (fv(M) - X) U fv(N) .

  *** Substitution equations
  eq X [N / X] = N .
  ceq Y [N / X] = Y if X =/= Y .
  eq (M N)[P / X] = (M [P / X])(N [P / X]) .
  eq (\ X . M)[N / X] = \ X . M .
  ceq (\ Y . M)[N / X] = \ Y . (M [N / X])
       if X =/= Y and (not(Y in fv(N)) or not(X in fv(M))) .
  ceq (\ Y . M)[N / X]
     = \ (new(fv(M N))) . ((M [new(fv(M N)) / Y])[N / X])
      if X =/= Y and (Y in fv(N)) and (X in fv(M)) .

  *** Alpha conversion
  ceq \ Y . (M [Y / X]) = \ X . M if not(Y in fv(M)) .
 endfm)

(mod BETA-ETA[X :: VAR] is
  including LAMBDA[X] .
  var X : Var.X .
  vars M N : Lambda[X] .
  rl [beta] : (\ X . M) N => M [N / X] .
  crl [eta] : \ X . (M X) => M if not(X in fv(M)) .
 endm)
```

We instantiate this parameterized module by using numbers for variables, where the new variable with respect to a given finite set is obtained as the maximum plus one. In order to be able to have the usual number notation, we import the built-in module `MACHINE-INT`.

First we instantiate the parameterized module for sets in order to get a module `EXT-MACHINE-INT` for sets of integers, where we also define the `max` operation. Then, we can define a view from the theory `VAR` to this module, where we make use of the feature that operations can be mapped to terms.

```
(view MachineInt from TRIV to MACHINE-INT is
  sort Elt to MachineInt .
 endv)

(fmod EXT-MACHINE-INT is
  pr (SET * (op _-_ to diff, op __ to _U_))[MachineInt] .
  op max : Set[MachineInt] -> MachineInt .
  var N : MachineInt .
  var S : Set[MachineInt] .
```



```
  eq max(empty-set) = 0 .
  eq max(N U S) = if N > max(S) then N else max(S) fi .
 endfm)

(view MachineInt' from VAR to EXT-MACHINE-INT is
  sort Var to MachineInt .
  sort VarSet to Set[MachineInt] .
  var S : VarSet .
  op new(S) to term max(S) + 1 .
  op _-_ to diff .
 endv)

(mod UNTYPED-LAMBDA-CALCULUS is
  protecting BETA-ETA[MachineInt'] .
 endm)
```

Reduction for untyped λ-calculus is confluent (the well-known Church-Rosser theorem), but not terminating. For example, the term

```
 (\ 1 . (1 1))(\ 1 . (1 1))
```

reduces to itself, as the following trace (that has been modified by hand to make it more readable) illustrates, where we ask for a sequence of only two rewrites:

```
Maude> set trace on .
Maude> trace exclude EXT-FULL-MAUDE .
Maude> set trace eqs off .
Maude> (rew-[2] (\ 1 . (1 1))(\ 1 . (1 1)) .)

*********** rule
rl [beta]: (\ X . M) N => M[N / X] .
M:Lambda`[MachineInt'`] --> 1 1
N:Lambda`[MachineInt'`] --> \ 1 . (1 1)
X:MachineInt --> 1
(\ 1 . (1 1)) \ 1 . (1 1)
--->
(1 1)[\ 1 . (1 1) / 1]
*********** rule
rl [beta]: (\ X . M) N => M[N / X] .
M:Lambda`[MachineInt'`] --> 1 1
N:Lambda`[MachineInt'`] --> \ 1 . (1 1)
X:MachineInt --> 1
(\ 1 . (1 1)) \ 1 . (1 1)
--->
```

```
(1 1)[\ 1 . (1 1) / 1]

rewrite in UNTYPED-LAMBDA-CALCULUS : ( \ 1 . ( 1 1 ) ) \ 1 . ( 1 1 ) .
result Lambda`[MachineInt'`] : ( \ 1 . ( 1 1 ) ) \ 1 . ( 1 1 )
Maude> set trace off .
```

The following trace corresponds to reduction to normal form, where no limit in the number of rewrites is required.

```
Maude> set trace on .
Maude> trace exclude EXT-FULL-MAUDE .
Maude> set trace eqs off .
Maude> (rew (\ 1 . ((1 (\ 2 . ((1 2) 2))) 1)) (\ 3 . (\ 4 . 3)) .)

*********** rule
rl [beta]: (\ X . M) N => M[N / X] .
M:Lambda`[MachineInt'`] --> (1 \ 2 . ((1 2) 2)) 1
N:Lambda`[MachineInt'`] --> \ 3 . \ 4 . 3
X:MachineInt --> 1
(\ 1 . ((1 \ 2 . ((1 2) 2)) 1)) \ 3 . \ 4 . 3
--->
((1 \ 2 . ((1 2) 2)) 1)[\ 3 . \ 4 . 3 / 1]
*********** rule
rl [beta]: (\ X . M) N => M[N / X] .
M:Lambda`[MachineInt'`] --> \ 4 . 3
N:Lambda`[MachineInt'`] --> \ 2 . (((\ 3 . \ 4 . 3) 2) 2)
X:MachineInt --> 3
(\ 3 . \ 4 . 3) \ 2 . (((\ 3 . \ 4 . 3) 2) 2)
--->
(\ 4 . 3)[\ 2 . (((\ 3 . \ 4 . 3) 2) 2) / 3]
*********** rule
rl [beta]: (\ X . M) N => M[N / X] .
M:Lambda`[MachineInt'`] --> \ 2 . (((\ 3 . \ 4 . 3) 2) 2)
N:Lambda`[MachineInt'`] --> \ 3 . \ 4 . 3
X:MachineInt --> 4
(\ 4 . \ 2 . (((\ 3 . \ 4 . 3) 2) 2)) \ 3 . \ 4 . 3
--->
(\ 2 . (((\ 3 . \ 4 . 3) 2) 2))[\ 3 . \ 4 . 3 / 4]
*********** trial #1
crl [eta]: \ X . (M X) => M if not X in fv(M) = true .
M:Lambda`[MachineInt'`] --> (\ 3 . \ 4 . 3) 2
X:MachineInt --> 2
*********** failure #1
*********** exhausted (#1)
```



```
*********** rule
rl [beta]: (\ X . M) N => M[N / X] .
M:Lambda`[MachineInt'`] --> \ 4 . 3
N:Lambda`[MachineInt'`] --> 2
X:MachineInt --> 3
(\ 3 . \ 4 . 3) 2
--->
(\ 4 . 3)[2 / 3]
*********** trial #2
crl [eta]: \ X . (M X) => M if not X in fv(M) = true .
M:Lambda`[MachineInt'`] --> \ 4 . 2
X:MachineInt --> 2
*********** failure #2
*********** exhausted (#2)
*********** rule
rl [beta]: (\ X . M) N => M[N / X] .
M:Lambda`[MachineInt'`] --> 2
N:Lambda`[MachineInt'`] --> 2
X:MachineInt --> 4
(\ 4 . 2) 2
--->
2[2 / 4]

rewrite in UNTYPED-LAMBDA-CALCULUS :
    ( \ 1 . ( ( 1 \ 2 . ( ( 1 2 ) 2 ) ) 1 ) ) \ 3 . \ 4 . 3 .
result Lambda`[MachineInt'`] : \ 2 . 2
Maude> set trace off .
```

Finally, consider the term

```
  (\ 1 . (\ 2 . 2))((\ 1. (1 1))(\ 1.(1 1)))
```

where the constant function `\ 1 . (\ 2 . 2)` that discards its argument is applied to the self-reducing term that we have seen above. Under eager evaluation, the reduction of this term does not terminate, since the argument is always reduced before applying the function. Under normal-order evaluation, the function is applied without reducing the argument, and thus reduction reaches in this case a normal form. Maude's top-down default strategy for evaluation of rules in system modules agrees with the latter, as the following shows:

```
Maude> (rew (\ 1 . (\ 2 . 2))((\ 1 . (1 1))(\ 1 .(1 1))) .)

*********** rule
rl [beta]: (\ X . M) N => M[N / X] .
```

```
M:Lambda`[MachineInt'`] --> \ 2 . 2
N:Lambda`[MachineInt'`] --> (\ 1 . (1 1)) \ 1 . (1 1)
X:MachineInt --> 1
(\ 1 . \ 2 . 2) ((\ 1 . (1 1)) \ 1 . (1 1))
--->
(\ 2 . 2)[(\ 1 . (1 1)) \ 1 . (1 1) / 1]

rewrite in UNTYPED-LAMBDA-CALCULUS :
    ( \ 1 . \ 2 . 2 ) ( ( \ 1 . ( 1 1 ) ) \ 1 . ( 1 1 ) ) .
result Lambda`[MachineInt'`] : \ 2 . 2
```

9 Concurrent object-oriented programming

The concurrent object-oriented programming paradigm can model in a very natural way concurrent interactions between objects in the real world. Rewriting logic supports a logical theory of concurrent objects that can be reduced to a particular class of rewrite theories.

9.1 Object-oriented systems

An *object* in a given state is represented as a term

```
< O : C | a1 : v1,..., an : vn >
```

where `O` is the object's name, belonging to a set `Oid` of object identifiers, `C` is its *class*, the `ai`'s are the names of the object's *attributes*, and the `vi`'s are their corresponding values. *Messages* are defined by the user for each application.

In a concurrent object-oriented system the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting (modulo the multiset structural axioms of associativity, commutativity, and identity) using rules that describe the effects of *communication events* between some objects and messages.

The following module `CONFIGURATION` defines the basic concepts of concurrent object systems. Note that the sorts `Msg` and `Attribute`, as well as the sorts `Oid` and `Cid` of object and class identifiers, are left unspecified. They will become fully defined when the `CONFIGURATION` module is extended by specific object-oriented definitions in a given object-oriented module.

```
fmod CONFIGURATION is
  sorts Oid Cid Attribute AttributeSet Object Msg Configuration .
  subsorts Object Msg < Configuration .
  subsort Attribute < AttributeSet .
```



```
  op none : -> AttributeSet [ctor] .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
                                      [ctor assoc comm id: none] .
  op <_:_| > : Oid Cid -> Object [ctor] .
  op <_:_|_> : Oid Cid AttributeSet -> Object [ctor] .
  op none : -> Configuration [ctor] .
  op __ : Configuration Configuration -> Configuration
                                       [ctor assoc comm id: none] .
endfm
```

In Full Maude, concurrent object-oriented systems can be defined by means of *object-oriented modules* introduced by the keyword `omod`, using a syntax more convenient than that of system modules because it assumes acquaintance with the basic entities, such as objects, messages and configurations, and supports linguistic distinctions appropriate for the object-oriented case. In particular, all object-oriented modules implicitly include the above `CONFIGURATION` module and assume its syntax. We can regard the special syntax reserved for object-oriented modules as syntactic sugar. In fact, each object-oriented module can be translated into a corresponding system module whose semantics *is* by definition that of the original object-oriented module.

The rewrite rules in the module specify in a declarative way the behavior associated with the messages. The general form of such rules is

$$\begin{array}{l} M_1 \ldots M_n \; \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle \\ \longrightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\ \qquad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\ \qquad M'_1 \ldots M'_q \\ \textit{if } C \end{array}$$

where $k, p, q \geq 0$, the M_s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is a rule condition. The result of applying a rewrite rule is that:

- the messages $M_1, \ldots, M_n$ disappear;
- the state and possibly the class of the objects $O_{i_1}, \ldots, O_{i_k}$ may change;
- all the other objects O_j vanish;
- new objects $Q_1, \ldots, Q_p$ are created;
- new messages $M'_1, \ldots, M'_q$ are sent.

Since the above rule involves several objects and messages in its lefthand side, we say that it is a *synchronous rule*.

It is conceptually important to distinguish the special case of rules involving at most one object and one message in their lefthand side. These rules are called *asynchronous* and have the form

$$\begin{array}{l} (M)\ \langle O : F \mid atts \rangle \\ \longrightarrow (\langle O : F' \mid atts' \rangle) \\ \quad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\ \quad M'_1 \ldots M'_q \\ if\ C \end{array}$$

where the notation (M) means that the message M is only an optional part of the lefthand side, that is, it is also possible to have *autonomous objects* that can act on their own without receiving any messages. Similarly, the notation $(\langle O : F' \mid atts' \rangle)$ means that the object O—in a possibly different state—is only an optional part of the righthand side, i.e., that it can be omitted in some rules so that the object is then deleted.

By convention, the only object attributes made explicit in a rule are those relevant for that rule. In particular, the attributes mentioned only in the lefthand side of the rule are preserved unchanged, the original values of attributes mentioned only in the righthand side of the rule do not matter, and all attributes not explicitly mentioned are left unchanged.

Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration `C < C'` in an object-oriented module is a particular case of a subsort declaration `C < C'`. The effect of a subclass declaration is that the attributes, messages, and rules of all the superclasses as well as the newly defined attributes, messages, and rules of the subclass characterize the structure and behavior of the objects in the subclass.

More details on Maude object-oriented systems can be found in [22, 13].

9.2 Bank accounts

The following object-oriented module specifies the concurrent behavior of objects in a class `Account` of bank accounts, each having a `bal`(ance) attribute, which may receive messages for crediting or debiting the account, or for transferring funds between two accounts.

Note that modules and commands are again enclosed in parentheses and some characters are preceded by a backquote "escape" character, because for object-oriented modules we are using the Full Maude extension.

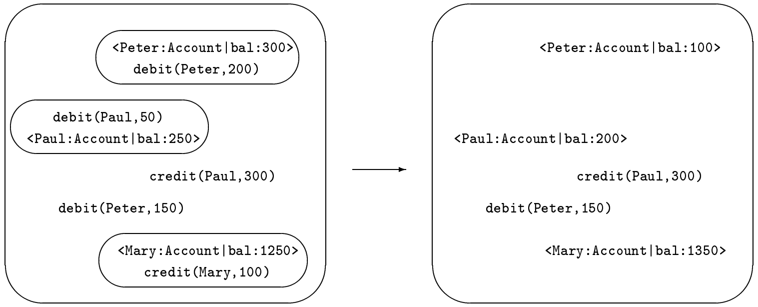


Figure 4: Concurrent rewriting of bank accounts.

```
(omod ACCOUNT is
   protecting MACHINE-INT .
   protecting QID .
   subsort Qid < Oid .
   class Account | bal : MachineInt .
   msgs credit debit : Oid MachineInt -> Msg .
   msg transfer_from_to_ : MachineInt Oid Oid -> Msg .
   vars A B : Oid .
   vars M N N' : MachineInt .

   rl [credit] : credit(A,M) < A : Account | bal : N >
               => < A : Account | bal : N + M > .

   crl [debit] : debit(A,M) < A : Account | bal : N >
               => < A : Account | bal : N - M >
               if  N >= M .

   crl [transfer] : (transfer M from A to B)
                    < A : Account | bal : N > < B : Account | bal : N' >
                  => < A : Account | bal : N - M >
                     < B : Account | bal : N' + M >
                  if  N >= M .
 endom)
```

Figure 4 shows graphically the (concurrent) application of the rewrite rules to the configuration on the lefthand side, obtaining as result the configuration on the righthand side, where further rewriting is still possible.

Suppose that now we want to define a subclass for checking accounts. A checking account has an additional attribute keeping the checking history represented as a lists of checks, that in turn are pairs of integers (check number and check amount). We also add a rule to cash a check in the account.

Note the use of the module constructor `TUPLE(2)` to create a module specifying pairs of integers. Since lists of such pairs are used to represent a checking history, sorts are appropriately renamed in the `protecting` importation below.

```
(view MachineInt from TRIV to MACHINE-INT is
   sort Elt to MachineInt .
 endv)

(view Tuple‘[MachineInt‘,MachineInt‘]
           from TRIV to TUPLE(2)[MachineInt, MachineInt] is
   sort Elt to Tuple[MachineInt, MachineInt] .
 endv)

(omod CHECKING-ACCOUNT is
   including ACCOUNT .

   pr LIST[Tuple‘[MachineInt‘,MachineInt‘]]
      * (sort List[Tuple‘[MachineInt‘,MachineInt‘]] to CheckHistory,
         sort Tuple[MachineInt, MachineInt] to Check,
         op  ‘(_‘,_‘) to <<_;_>>) .

   class ChkAccount | chk-hist : CheckHistory .
   subclass ChkAccount < Account .
   msg chk_#_amt_ : Oid MachineInt MachineInt -> Msg .

   var A : Oid .
   vars K M N : MachineInt .
   var H : CheckHistory .
   crl [cash] : (chk A # K amt M)
                 < A : ChkAccount | bal : N, chk-hist : H >
              => < A : ChkAccount | bal : (N - M),
                                    chk-hist : (H ++ (<< K ; M >> : [])) >
              if N > M .
 endom)

Maude> (rew < 'Paul : ChkAccount | bal : 5000, chk-hist : [] >
            < 'Peter : Account | bal : 2000 >
            < 'Mary : ChkAccount | bal : 9000, chk-hist : [] >
            debit('Peter, 1000) (chk 'Paul # 123 amt 800)
            credit('Paul, 1300) credit('Mary, 200) .)
```


2	8	3
1		4
7	6	5

1	2	3
8		4
7	6	5

Figure 5: Initial and final states for puzzle.

```
result Configuration :
    < 'Peter : Account | bal : 1000 >
    < 'Paul : ChkAccount | bal : 5500 , chk-hist : << 123 ; 800 >> : [ ] >
    < 'Mary : ChkAccount | bal : 9200 , chk-hist : [ ] >
```

9.3 A puzzle

We show in this section an object-oriented approach to another typical artificial intelligence planning problem, the 8-puzzle problem as presented in [15, p. 283]. There is a set of eight numbered tiles and a blank tile, arranged in a 3×3-grid. By moving the blank tile up, down, left, or right, the goal is to reach the state pictured on the righthand side of Figure 5, starting in an arbitrary state, like for example the one pictured on the lefthand side.

Each tile is represented as an object `< (R,C) : Tile | value : N >`, where `R` and `C` denote the row and column, respectively, and `N` denotes either a natural number between 1 and 8, or the marker `blank`. We want useful coordinates to range between 1 and 3, and number values to range between 1 and 8, thus there are some ad-hoc overloaded constants. For the values there is no operation, while we need to check adyacent tiles using an auxiliary successor function `s_` on the sort `Coordinate`. Movements are represented as messages sent to the tile configuration.

```
(omod 8-PUZZLE is
   sorts NumValue Value Coordinate .
   subsort NumValue < Value .
   ops 1 2 3 4 : -> Coordinate [ctor] .
   ops 1 2 3 4 5 6 7 8 : -> NumValue [ctor] .
   op blank : -> Value [ctor] .
   op s_ : Coordinate -> Coordinate .
   eq s 1 = 2 .
   eq s 2 = 3 .
   eq s 3 = 4 .
   eq s 4 = 4 .
   op ‘(_‘,_‘) : Coordinate Coordinate -> Oid .
   class Tile | value : Value .
```

```
msgs left right up down : -> Msg .
vars N M R : Coordinate .
var P : NumValue .

crl [l] : left < (N,R) : Tile | value : blank >
               < (N,M) : Tile | value : P >
        => < (N,R) : Tile | value : P >
           < (N,M) : Tile | value : blank >
        if R == s M .

crl [r] : right < (N,R) : Tile | value : blank >
                < (N,M) : Tile | value : P >
        => < (N,R) : Tile | value : P >
           < (N,M) : Tile | value : blank >
        if s R == M .

crl [u] : up < (R,M) : Tile | value : blank >
             < (N,M) : Tile | value : P >
        => < (R,M) : Tile | value : P >
           < (N,M) : Tile | value : blank >
        if s R == N .

crl [d] : down < (R,M) : Tile | value : blank >
               < (N,M) : Tile | value : P >
        => < (R, M) : Tile | value : P >
           < (N,M) : Tile | value : blank >
        if R == s N .
endom)
```

The final state in Figure 5 is represented by the following tile configuration:

```
< (1,3) : Tile | value : 1 > < (2,3) : Tile | value : 2 >
< (3,3) : Tile | value : 3 > < (1,2) : Tile | value : 8 >
< (2,2) : Tile | value : blank > < (3,2) : Tile | value : 4 >
< (1,1) : Tile | value : 7 > < (2,1) : Tile | value : 6 >
< (3,1) : Tile | value : 5 >
```

A plan that solves the planning problem in Figure 5 is (in a self-explanatory intuitive notation) `up; left; down; right`. However, one must be aware that this is a *sequential* plan, while messages in the tile configuration do not have any order upon them because of the structural axioms for multisets. Therefore, this is another example where strategies are useful to control the application of rewrite rules in the order wanted by the user.

9.4 A simple spreadsheet

The following object-oriented module specifies the concurrent behavior of objects in a simple class `Cell` of cells in a spreadsheet, whose unique attribute is the value stored in the cell. The cells are organized in a grid and are therefore identified by means of pairs `(N,M)` giving the row and column numbers. For each row `N` there is a cell `(N,total)` that keeps track of the corresponding total, and similarly for each column `M` there is a cell `(total,M)`. There is also a cell `(total,total)` providing the sum of all the values in all the cells in the spreadsheet. The spreadsheet may receive messages `add(N,M,V)` and `sub(N,M,V)` for adding or subtracting the amount `V` to the value stored in cell `(N,M)`.

```
(omod SPREADSHEET is
   protecting NAT .
   sort Name .
   subsort Nat < Name .
   op total : -> Name [ctor] .
   op `(_`,_`) : Name Name -> Oid .
   class Cell | val : Nat .
   msgs add sub : Nat Nat Nat -> Msg .
   vars M N V W X Y Z : Nat .

   rl [add] : add(N,M,V) < (N,M) : Cell | val : W >
                         < (total,total) : Cell | val : X >
                         < (N,total) : Cell | val : Y >
                         < (total,M) : Cell | val : Z >
           => < (N,M) : Cell | val : W + V >
              < (total,total) : Cell | val : X + V >
              < (N,total) : Cell | val : Y + V >
              < (total,M) : Cell | val : Z + V > .

   crl [sub] : sub(N,M,V) < (N,M) : Cell | val : W >
                          < (total,total) : Cell | val : X >
                          < (N,total) : Cell | val : Y >
                          < (total,M) : Cell | val : Z >
            => < (N,M) : Cell | val : W - V >
               < (total,total) : Cell | val : X - V >
               < (N,total) : Cell | val : Y - V >
               < (total,M) : Cell | val : Z - V >
            if V <= W .
endom)
```

We show here how to transform synchronous object-oriented rules like the ones in the module above, into asynchronous rules where there is only one object in the rule's lefthand side, as we have described in Section 9.1. The essential idea is to introduce new messages in the righthand side of the rules, creating new states in which the original computation is half-done, and is going to continue by further interaction of the new messages with the objects.

```
(omod SPREADSHEET-ASYNCH is
   protecting NAT .
   sort Name .
   subsort Nat < Name .
   op total : -> Name [ctor] .
   op `(_`,_`) : Name Name -> Oid .
   class Cell | val : Nat .
   msgs add sub : Nat Nat Nat -> Msg .
   msgs add-row add-col : Nat Nat -> Msg .
   msgs sub-row sub-col : Nat Nat -> Msg .
   msgs add-total sub-total : Nat -> Msg .
   vars M N V W : Nat .

   rl [add] : add(N,M,V) < (N,M) : Cell | val : W >
            => < (N,M) : Cell | val : W + V >
               add-row(N,V) add-col(M,V) add-total(V) .
   rl [add-row] : add-row(N,V) < (N,total) : Cell | val : W >
                => < (N,total) : Cell | val : W + V > .
   rl [add-col] : add-col(M,V) < (total,M) : Cell | val : W >
                => < (total,M) : Cell | val : W + V > .
   rl [add-total] : add-total(V) < (total,total) : Cell | val : W >
                  => < (total,total) : Cell | val : W + V > .

   crl [sub] : sub(N,M,V) < (N,M) : Cell | val : W >
             => < (N,M) : Cell | val : W - V >
                sub-row(N,V) sub-col(M,V) sub-total(V)
             if V <= W .
   rl [sub-row] : sub-row(N,V) < (N,total) : Cell | val : W >
                => < (N,total) : Cell | val : W - V > .
   rl [sub-col] : sub-col(M,V) < (total,M) : Cell | val : W >
                => < (total,M) : Cell | val : W - V > .
   rl [sub-total] : sub-total(V) < (total,total) : Cell | val : W >
                  => < (total,total) : Cell | val : W - V > .
 endom)
```

Because of the presence of new messages, there are configurations in the module SPREADSHEET-ASYNCH that do not correspond to any configuration in the original module SPREADSHEET. However, in any computation using the new rules that starts

in a configuration from SPREADSHEET the new messages are going to eventually disappear by application of the new rules involving those messages on the lefthand side, reaching in this way a configuration in SPREADSHEET. Moreover, this configuration is exactly the same achieved by the original synchronous rules.

9.5 Blocks world

This is an object-oriented approach to a simpler version of the blocks world described in Section 8.4. Here, we have removed the robot arm to move blocks. A block is represented as an object with two attributes, `under`, saying whether it is under another block or it is clear, and `on`, saying whether the block is on top of another block or is on the table. Actions are represented as messages. The reader can compare this version with the one described later in Section 11.

```
(omod OO-BLOCKS-WORLD is
   protecting QID .
   sorts BlockId Up Down .
   subsorts Qid < BlockId < Oid .
   subsorts BlockId < Up Down .
   op clear : -> Up [ctor] .
   op table : -> Down [ctor] .
   class Block | under : Up, on : Down .
   msg move : Oid Oid Oid -> Msg .
   msgs unstack stack : Oid Oid -> Msg .
   vars X Y Z : BlockId .

   rl [move] : move(X,Z,Y) < X : Block | under : clear, on : Z >
               < Z : Block | under : X > < Y : Block | under : clear >
             => < X : Block | on : Y >
                < Z : Block | under : clear > < Y : Block | under : X > .

   rl [unstack] : unstack(X,Z) < X : Block | under : clear, on : Z >
                  < Z : Block | under : X >
             => < X : Block | on : table > < Z : Block | under : clear > .

   rl [stack] : stack(X,Z) < X : Block | under : clear, on : table >
                < Z : Block | under : clear >
              => < X : Block | on : Z > < Z : Block | under : X > .
 endom)
```

The states I and F in Figure 3 are respectively described by the following two configurations:

```
  < 'a : Block | under : 'c, on : table >
  < 'c : Block | under : clear, on : 'a >
  < 'b : Block | under : clear, on : table >

  < 'c : Block | under : clear, on : 'b >
  < 'b : Block | under : 'c, on : table >
  < 'a : Block | under : clear, on : table >
```

```
Maude> (rew < 'a : Block | under : 'c, on : table >
            < 'c : Block | under : clear, on : 'a >
            < 'b : Block | under : clear, on : table >
            unstack('c, 'a) .)
Result Configuration : < 'a : Block | on : table , under : clear >
                       < 'c : Block | on : table , under : clear >
                       < 'b : Block | on : table , under : clear >
```

Suppose that the blocks world is further refined so that now blocks can have colors, say red, blue, and yellow. Of course, we want the rules for manipulating blocks to remain "exactly as before." This is trivially achieved by class inheritance as illustrated by the following module, because the rules given for the class `Block` of blocks also apply without changes to blocks in the subclass `ColoredBlock` of colored blocks.

```
(omod OO-BLOCKS-WORLD+COLOR is
   including OO-BLOCKS-WORLD .
   sort Color .
   ops red blue yellow : -> Color [ctor] .
   class ColoredBlock | color : Color .
   subclass ColoredBlock < Block .
 endom)
```

```
Maude> (rew < 'a : Block | color : red, under : 'c, on : table >
            < 'c : Block | color : blue, under : clear, on : 'a >
            < 'b : Block | color : yellow, under : clear, on : table >
            unstack('c, 'a) .)
Result Configuration :
       < 'a : Block | on : table , under : clear , color : red >
       < 'c : Block | on : table , under : clear , color : blue >
       < 'b : Block | on : table , under : clear , color : yellow >
```

9.6 Stacks as linked objects

In this section we describe an object-oriented parameterized module defining a stack of elements. We define a class `Stack[X]` as a linked sequence of node objects.

Objects of class `Stack[X]` only have an attribute `first`, containing the identifier of the first node in the stack. If the stack is empty the value associated to the `first` attribute is `null`.

Each object of class `Node[X]` has an attribute `contents` to store a value of sort `Elt.X`, and an attribute `next` holding the identifier of the next node, which will be `null` if there is no next node. Notice that the identifiers of the nodes are of the form `o(S,N)`, where `S` is the identifier of the stack object to which the node belongs, and `N` is a natural number.

The messages `push`, `pop`, and `top` have as their first argument the identifier of the object to which they are addressed, and will cause, respectively, the insertion at the top of the stack of a new element, the deletion of the top element, and the sending of a response message `elt` containing the element at the top of the stack to the object making the request.

```
(omod STACK1[X :: TRIV] is
   protecting MACHINE-INT .
   protecting QID .
   subsort Qid < Oid .
   class Node[X] | next : Oid, contents : Elt.X .
   class Stack[X] | first : Oid .
   msg _push_ : Oid Elt.X -> Msg .
   msg _pop : Oid -> Msg .
   msg _top_ : Oid Oid -> Msg .
   msg _elt_ : Oid Elt.X -> Msg .

   op null : -> Oid .
   op o : Oid MachineInt -> Oid .

   vars O O' O'' : Oid .
   var E : Elt.X .
   var N : MachineInt .

   rl [top] : *** top on a nonempty stack
              < O : Stack[X] | first : O' >
              < O' : Node[X] | contents : E > (O top O'')
            => < O : Stack[X] | > < O' : Node[X] | > (O'' elt E) .

   rl [push1] : *** push on a nonempty stack
                < O : Stack[X] | first : o(O, N) > (O push E)
              => < O : Stack[X] | first : o(O, N + 1) >
                < o(O, N + 1) : Node[X] | contents : E, next : o(O, N) > .

   rl [push2] : *** push on an empty stack
```

```
          < O : Stack[X] | first : null > (O push E)
        => < O : Stack[X] | first : o(O, 0) >
           < o(O, 0) : Node[X] | contents : E, next : null > .

   rl [pop] : *** pop on a nonempty stack
         < O : Stack[X] | first : O' > < O' : Node[X] | next : O'' >
         (O pop)
       => < O : Stack[X] | first : O'' > .
 endom)
```

We may also define stacks not storing data elements of a particular sort, but actually objects in a particular class. We can define an object-oriented module with the intended behavior as a parameterized module STACK2 parameterized by an object-oriented theory CELL as follows:

```
(oth CELL is
   sort Elt .
   class Cell | contents : Elt .
 endoth)

(omod STACK2[X :: CELL] is
   protecting MACHINE-INT .
   protecting QID .
   subsort Qid < Oid .
   class Node[X] | next : Oid, node : Oid .
   class Stack[X] | first : Oid .
   msg _push_ : Oid Oid -> Msg .
   msg _pop : Oid -> Msg .
   msg _top_ : Oid Oid -> Msg .
   msg _elt_ : Oid Elt.X -> Msg .

   op null : -> Oid .
   op o : Oid MachineInt -> Oid .

   vars O O' O'' O''' : Oid .
   var E : Elt.X .
   var N : MachineInt .

   rl [top] : *** top on a nonempty stack
         < O : Stack[X] | first : O' > < O' : Node[X] | node : O'' >
         < O'' : Cell.X | contents : E > (O top O''')
       => < O : Stack[X] | > < O' : Node[X] | >
          < O'' : Cell.X | > (O''' elt E) .
```



```
   rl [push1] : *** push on a nonempty stack
           < O : Stack[X] | first : o(O, N) > (O push O')
         => < O : Stack[X] | first : o(O, N + 1) >
            < o(O, N + 1) : Node[X] | next : o(O, N), node : O' > .

   rl [push2] : *** push on an empty stack
           < O : Stack[X] | first : null > (O push O')
         => < O : Stack[X] | first : o(O, 0) >
            < o(O, 0) : Node[X] | next : null, node : O' > .

   rl [pop] : *** pop on a nonempty stack
         < O : Stack[X] | first : O' > < O' : Node[X] | next : O'' >
         (O pop)
       => < O : Stack[X] | first : O'' > .
 endom)

(view Account from CELL to ACCOUNT is
   sort Elt to MachineInt .
   class Cell to Account .
   attr contents . Cell to bal .
 endv)

(omod ACCOUNT-STACK is
   protecting STACK2[Account] .
 endom)
```

10 Reflection and metaprogramming

Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective [10, 4] in the specific sense that its metalevel can be represented at the object level by a *universal theory* that can represent all other theories, including itself. More precisely, 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

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

This makes the metatheory of rewriting logic accessible to the user in a clear and principled way, and makes possible many advanced metaprogramming applications, including user-definable strategy languages, language extensions by new module composition operations, development of theorem proving tools, and reifications of other languages and logics within rewriting logic [9].

10.1 The META-LEVEL module

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module META-LEVEL. Furthermore, several other useful functions are also built-in for efficiency reasons. In the module META-LEVEL:

- Maude terms are reified as elements of a data type Term of terms;
- Maude modules are reified as terms in a data type Module of modules;
- the processes of reducing a term to normal form in a functional module and of finding whether such a normal form has a given sort are reified by a function meta-reduce;
- the process of applying a rule of a system module to a subject term is reified by a function meta-apply;
- the process of rewriting a term in a system module using Maude's default interpreter is reified by a function meta-rewrite; and
- parsing and pretty printing of a term in a signature, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also reified by corresponding metalevel functions.

In this section we review the most important features provided by this module and refer the reader to [5, 6] for more details.

10.1.1 Representing terms

Terms are reified as elements of the data type Term of terms, by means of the following operations:

```
subsort Qid < Term .
subsort Term < TermList .
op {_}_ : Qid Qid -> Term [ctor] .
op _[_] : Qid TermList -> Term [ctor] .
op _,_ : TermList TermList -> TermList [ctor assoc] .
op error* : -> Term .
```

The first declaration, making Qid a subsort of Term, is used to represent variables by the corresponding quoted identifiers. Thus, the variable N is represented by 'N. The operation {_}_ is used for representing constants as pairs, with the first argument the constant in quoted form, and the second argument the sort of the constant, also in quoted form. For example, the constant 0 of sort Nat in the module BASIC-NAT in Section 2.4 (or any of its extensions) is represented as {'0}'Nat. The

operation _[_] corresponds to the recursive construction of terms out of subterms, with the first argument the top operation in quoted form, and the second argument the list of its subterms, where list concatenation is denoted _,_. For example, the term s(s(0)) + s(0) of sort Nat in module NAT is metarepresented as the following term of sort Term in module META-LEVEL:

```
'_+_['s['s[{'0}'Nat]], 's[{'0}'Nat]]}
```

Essentially, terms are written in a prefix form, all identifiers are quoted, constants are annotated with the corresponding sort, and parentheses become square brackets.

Since terms in the module META-LEVEL can be metarepresented just as terms in any other module, the representation of terms can be iterated. For example, the meta-metarepresentation $\overline{\overline{\texttt{s}(0)}}$ of the term s(0) in NAT is the term

```
'_`[_`][{''s}'Qid,'`{_`}_[{''0}'Qid,{''Nat}'Qid]]
```

The last declaration above for the data type of terms is a constant error* to be used as an error element.

10.1.2 Representing modules

Functional and system modules are metarepresented in a syntax very similar to their original user syntax. The main differences are that: (1) terms in equations, membership axioms, and rules are now metarepresented as we have already explained; (2) in the metarepresentation of modules we follow a *fixed order* in introducing the different kinds of declarations for sorts, subsorts, variables, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way; and (3) sets of identifiers—used in declarations of sorts—are represented as sets of quoted identifiers built with an associative and commutative operation _;_.

The module META-LEVEL provides sorts and constructors for each of the declarations that can appear in modules. For example, we have sorts OpDecl and OpDeclSet to represent declarations of operations and sets of declarations of operations, respectively. If there are no operation declarations in a particular module, then this argument will be none. Otherwise, the set is given by the associative and commutative constructor __, which is declared with identity element none. Each of the operation declarations is then given by a term of sort OpDecl using the constructor

```
op _:_->_[_]. : Qid QidList Qid AttrSet -> OpDecl [ctor] .
```

The last argument of this constructor corresponds to the set of attributes of an operation.

The `META-LEVEL` syntax for the top-level operations representing functional and system modules is as follows:

```
  op fmod_is_______endfm : Qid ImportList SortDecl
        SubsortDeclSet OpDeclSet
        VarDeclSet MembAxSet EquationSet -> Module [ctor] .

  op mod_is________endm : Qid ImportList SortDecl
        SubsortDeclSet OpDeclSet
        VarDeclSet MembAxSet EquationSet RuleSet -> Module [ctor] .
```

To illustrate the general syntax for representing modules, we use yet another module `YANAT` for natural numbers with zero, successor, and commutative addition. We first give `YANAT`'s ordinary Maude syntax:

```
fmod YANAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
endfm
```

The representation $\overline{\texttt{YANAT}}$ of `YANAT` as a term of sort `Module` in `META-LEVEL` is

```
fmod 'YANAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [ctor] .
  op 's : 'Nat -> 'Nat [ctor] .
  op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
  var 'N : 'Nat .
  var 'M : 'Nat .
  none
  eq '_+_[{'0}'Nat, 'N] = 'N .
  eq '_+_['s['N], 'M] = 's['_+_['N, 'M]] .
endfm
```


Since `YANAT` has no list of imported submodules and no membership axioms, those fields are filled by the constant `nil` of sort `ImportList`, and the constant `none` of sort `MembAxSet`, respectively. Similarly, `none` is the empty set of attributes for operations that have no attributes.

Note that—just as in the case of terms—terms of sort `Module` can be metarepresented again, yielding then a term of sort `Term`, and this can be iterated an arbitrary number of times. This is in fact necessary when a metalevel computation has to operate at higher levels, like, for example, in some theorem proving tools described in [9], where modules are metarepresented as terms in inference rules, but they have to be meta-metarepresented as terms of sort `Term` when used in strategies that control the application of the inference rules.

10.1.3 Descent functions

The module `META-LEVEL` has three built-in operations for meta-evaluation, also called *descent functions* [5], `meta-reduce`, `meta-rewrite`, and `meta-apply`, that provide three useful and efficient ways of reducing metalevel computations to object-level ones.

The operation `meta-reduce` has the following declaration, where we have omitted the lengthy list of bindings to C++ code in its declaration as a `special` built-in operation:

```
  op meta-reduce : Module Term -> Term [special (...) ] .
```

It takes as arguments the representations of a module $\mathcal{R}$ and of a term t in that module, and returns the representation of the canonical form of the term t using the equations in $\mathcal{R}$, e.g,

```
Maude> red meta-reduce(YANAT, s(0) + 0) .
result Term: s(0)
```

The operation `meta-rewrite` is declared as

```
  op meta-rewrite : Module Term MachineInt -> Term [special (...) ] .
```

It is entirely analogous to `meta-reduce`, but instead of using only the equational part of a module it now uses both the equations and the rules to rewrite the term using Maude's default strategy. Its first two arguments are the representations of a module $\mathcal{R}$ and of a term t, and its third argument is a positive machine integer n. Its result is the representation of the term obtained from t after at most n applications of the rules in $\mathcal{R}$ using the strategy of Maude's default interpreter. When the value 0 is given as the third argument, no bound is given to the number of rewrites, and rewriting proceeds to the bitter end.

The operation `meta-apply` is declared as

```
op meta-apply : Module Term Qid Substitution
                MachineInt -> ResultPair [special (...) ] .
```

The first four arguments are representations in `META-LEVEL` of a module $\mathcal{R}$, a term t in $\mathcal{R}$, a label l of some rules[1] in $\mathcal{R}$, and a set of assignments (possibly empty) defining a partial substitution σ for the variables in those rules. The last argument is a natural number n. `meta-apply` then returns a pair, of sort `ResultPair`, consisting of a term and a substitution. The operation `meta-apply` is evaluated as follows:

1. the term t is first fully reduced using the equations in $\mathcal{R}$;
2. the resulting term is matched (at the top, with extension) against all rules with label l partially instantiated with σ, with matches that fail to satisfy the condition of their rule discarded;
3. the first n successful matches are discarded; if there is an $(n+1)$th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise the pair `{error*, none}` is returned, where `none` denotes the identity substitution;
4. the term resulting from applying the given rule with the $(n+1)$th match is fully reduced using the equations in $\mathcal{R}$;
5. the pair formed using the constructor `{_,_}` whose first element is the representation of the resulting fully reduced term and whose second element is the representation of the match used in the reduction is returned.

10.1.4 Changing reflection levels

Note that the use of the descent functions presented in the previous section requires giving as argument the metarepresentation of terms and modules, which can be very inconvenient in some cases. That is, what we sometimes write using the more intuitive bar notation to improve readability must in fact be substituted by the corresponding term. For example, the reduction given in Section 10.1.3 should in fact be given as

```
Maude> red meta-reduce(YANAT, '_+_['s[{'0}'Zero], {'0}'Zero]) .
result Term: 's[{'0}'Zero]
```

where we have used the fact that Maude allows just using the name of the module instead of its metarepresentation if it has been entered previously; otherwise, the

[1] A user can declare several rules with the same label; then `meta-apply` will try to apply some rule with that label.

metarepresentation of the module in which the term is to be reduced must also be given.

When using Full Maude we have available some additional facilities. For example, we can use the `up` function to avoid the cumbersome task of explicitly writing the metarepresentation of a term or the metarepresentation of a module. For example, to obtain the metarepresentation of a term such as `s(0)` in the module `YANAT` above, denoted $\overline{\texttt{s(0)}}$, we can write

```
Maude> (red up(YANAT, s(0)) .)
result Term : 's[{'0}'Zero]
```

Thus, instead of explicitly writing the metarepresentation $\overline{\texttt{s(0) + 0}}$ in the above reduction we can also write

```
Maude> (red meta-reduce(YANAT, up(YANAT, s(0) + 0)) .)
result Term : 's[{'0}'Zero]
```

Note that the module name is the first argument of the `up` function, with the term of that module to be metarepresented as the second argument. Since the same term can be parsed in different ways in different modules, and therefore can have different metarepresentations depending on the module in which it is considered, the module to which the term belongs has to be used in order to obtain the correct metarepresentation. Note also that the above reduction only makes sense at the metalevel, that is, in a module importing the module `META-LEVEL`.

The `up` function also provides us with a way of accessing the metarepresentation of a module. In any module importing the module `META-LEVEL` we can evaluate the `up` function with the name of any module previously entered as an argument, and then obtain as result the metarepresentation of such a module. Thus, assuming that the previous module `YANAT` has been entered in Full Maude, we can get its metarepresentation, denoted $\overline{\texttt{YANAT}}$, by evaluating in `META-LEVEL`, or in any other module importing `META-LEVEL`, the following command:

```
Maude> (red up(YANAT) .)
result FModule :
     fmod 'YANAT is
        nil
        sorts 'Zero ; 'Nat .
        subsort 'Zero < 'Nat .
        op '0 : nil -> 'Zero [ctor] .
        op 's : 'Nat -> 'Nat [ctor] .
        op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
        var 'M : 'Nat .
        var 'N : 'Nat .
```

```
    none
    eq '_+_[{'0}'Zero, 'N] = 'N .
    eq '_+_['s['N], 'M] = 's['_+_['N, 'M]] .
  endfm
```

This facility can be used to write reductions of terms as those presented in Section 10.1.3, for example of `meta-reduce(`$\overline{\texttt{YANAT}}$`,` $\overline{\texttt{s(s(0)) + s(s(s(0)))}}$`)`, as follows:

```
Maude> (red meta-reduce(up(YANAT), up(YANAT, s(s(0)) + s(s(s(0))))) .)
result Term : 's['s['s['s['s[{'0}'Zero]]]]]
```

The result of a metalevel computation that may use several levels of reflection can be a term or module metarepresented one or more times, which may be hard to read. Therefore, to display the output in a more readable form we can use the `down` command, which is in a sense inverse to `up`, since it gives us back the term from its metarepresentation. The `down` command takes two arguments: The first argument is the name of the module to which the term to be returned belongs; the metarepresentation of the desired output term should be the result of the command given as second argument. Thus, we can give the following command:

```
Maude> (down YANAT : meta-reduce(YANAT, up(YANAT, 0 + s(0))) .)
result Nat : s(0)
```

Notice that this is equivalent to what we wrote in Section 10.1.3 as

```
Maude> red meta-reduce(YANAT, s(0) + 0) .
result Term: s(0)
```

(with $\overline{\texttt{YANAT}}$, $\overline{\texttt{s(0) + 0}}$ and $\overline{\texttt{s(0)}}$ overlined as printed)

The use of `up` and `down` in Full Maude can be iterated with as many levels of reflection as we wish.

10.2 Metaprogramming

To illustrate some of the metaprogramming possibilities of Maude, we present the specification of a function that generates a test set, that is, a collection of pattern terms such that every ground term in a given sort is provably equal to an instance of one of those patterns. For example, `0` and `s(N)`, with `N` a variable of sort `Nat`, constitute a test set for the sort `Nat` of the natural numbers. Such test sets are used in inductive theorem proving and in other formal reasoning applications. To generate test sets we take the metarepresentation of a module as input, which we manipulate as data. This is possible in Maude thanks to its reflective capabilities.

Of course, the choice of which terms to use for a test set is not unique. For example, in a specification of the natural numbers with sorts `Nat` of natural numbers and `NzNat` of nonzero natural numbers, and constructors `0 : -> Nat` and `s_ : Nat -> NzNat`, we could use as test set for the sort `NzNat` the term `s N` with `N` a variable of sort `Nat`, but we could just as well have used the terms `s 0` and `s s N`, with `N` a variable of sort `Nat`.

Our test set generation function chooses the first, simpler alternative, that is, it always yields terms of depth zero or one in the test set of each sort. Such terms are obtained by inspecting which constructors have sort smaller or equal to the sort in question; for each such constructor $\texttt{f} : \texttt{s}_1 \ldots \texttt{s}_n \to \texttt{s}$ a term $\texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_n)$ with $\texttt{x}_i$: $\texttt{s}_i$, for $1 \leq i \leq n$, is then added to the test set.

Given a subsignature of constructors for which sufficient generation has been proved [8, 7], the function `genTestSets` takes a functional module and generates the test sets for each sort in it. Its declaration is as follows:

```
op genTestSets : FModule -> TestSetSolution .
```

where `TestSetSolution` is a new sort with constructor:

```
op <_;_> : FModule Set[Test] -> TestSetSolution [ctor] .
```

That is, the function returns a pair consisting of a module and the family of test sets that have been generated. Note that it may be that in the process of generating the test sets new variables have to be added to the original specification. Therefore, the test sets generated have to be used together with this new module.

To be able to distinguish the constructors in the module passed as argument to `genTestSets`, we make use of the attribute `ctor`. We assume that the operations with this attribute are constructors that generate all the data in sorts, that is, any ground term having a sort is provably equal to a constructor term.

We begin with the specification `TEST`, in which we define tests as pairs composed by a sort name, of sort `Qid`, and a set of terms. We use the parameterized specification for sets described in Section 7.2.

```
(view Term from TRIV to META-LEVEL is
   sort Elt to Term .
 endv)

(fmod TEST is
   pr QID .
   pr (SET * (op __ to termSet))[Term] .
   sort Test .
   op test : Qid Set[Term] -> Test [ctor] .
 endfm)
```

A test set is then given as a set of tests, and, as pointed out above, a test set of this form is given as part of the result of the function `genTestSets`. There are several auxiliary functions, which give the test set for each sort in the specification given as input. In particular, `test2` generates the set of terms as indicated above for a particular sort, given a set of declarations of variables and a set of declarations of operations, and `test3` gives a list of variables to be used as arguments in a term, given a list of sorts and a set of variables. Since we need to generate new variables in case there are not enough variables in the original module, we generate new variables by keeping an index with which to create them. `test2`, `test3`, and `test4`, in addition to returning a set of terms or a list of terms, return this index and the set of new variables. We return these values as triples with the appropriate declarations.

```
(view Test from TRIV to TEST is
   sort Elt to Test .
 endv)

(view Index from TRIV to MACHINE-INT is
   sort Elt to MachineInt .
 endv)

(view VarDeclSet from TRIV to META-LEVEL is
   sort Elt to VarDeclSet .
 endv)

(view Set`[Term`] from TRIV to (SET * (op __ to termSet))[Term] is
   sort Elt to Set[Term] .
 endv)

(view TermList from TRIV to META-LEVEL is
   sort Elt to TermList .
 endv)

(fmod TEST-SETS is
   pr META-LEVEL .
   pr TUPLE(3)[Index, VarDeclSet, Set`[Term`]] .
   pr TUPLE(3)[Index, VarDeclSet, TermList] * (op `(_`,_`,_`) to <_;_;_>) .
   pr SET[Test] .

   sort TestSetSolution .
   op <_;_> : FModule Set[Test] -> TestSetSolution [ctor] .

   vars VDS VDS' : VarDeclSet .
```



```
vars QI X S S' F : Qid .
var SL : QidList .
var SS : QidSet .
var IL : ImportList .
var SD : SortDecl .
var SSDS : SubsortDeclSet .
var ODS : OpDeclSet .
var MAS : MembAxSet .
var ES : EquationSet .
var M : FModule .
var N : MachineInt .
var TestS : Set[Test] .
var TS : Set[Term] .
var TL : TermList .
vars A A' : Attr .
var AS : AttrSet .

*** Auxiliary functions

op isThereVariable? : VarDeclSet Qid -> Bool .
eq isThereVariable?(((var X : S .) VDS), S')
  = (S == S') or isThereVariable?(VDS, S') .
eq isThereVariable?(none, S) = false .

op varDeclSet : FModule -> VarDeclSet .
eq varDeclSet(fmod QI is IL SD SSDS ODS VDS MAS ES endfm) = VDS .

op opDeclSet : FModule -> OpDeclSet .
eq opDeclSet(fmod QI is IL SD SSDS ODS VDS MAS ES endfm) = ODS .

op sortSet : FModule -> QidSet .
eq sortSet(fmod QI is IL sorts SS . SSDS ODS VDS MAS ES endfm) = SS .

op set : FModule VarDeclSet -> FModule .
eq set(fmod QI is IL SD SSDS ODS VDS MAS ES endfm, VDS')
  = fmod QI is IL SD SSDS ODS VDS' MAS ES endfm .

op _in_ : Attr AttrSet -> Bool .
eq A in (A' AS) = (A == A') or (A in AS) .
eq A in none = false .

*** Test Set Generation

op genTestSets : FModule -> TestSetSolution .
```

```
op genTestSets : FModule QidSet MachineInt VarDeclSet Set[Test]
      -> TestSetSolution .
op test2 : FModule Qid MachineInt OpDeclSet VarDeclSet Set[Term]
      -> Tuple[Index, VarDeclSet, Set`[Term`]] .
op test3 : MachineInt QidList VarDeclSet
      -> Tuple[Index, VarDeclSet, TermList] .
op test4 : MachineInt QidList VarDeclSet TermList VarDeclSet
      -> Tuple[Index, VarDeclSet, TermList] .

eq genTestSets(M)
  = genTestSets(M, sortSet(M), 0, varDeclSet(M), empty-set) .
eq genTestSets(M, (S ; SS), N, VDS, TestS)
  = genTestSets(M, SS,
       p1(test2(M, S, N, opDeclSet(M), VDS, empty-set)),
       p2(test2(M, S, N, opDeclSet(M), VDS, empty-set)),
       (test(S, p3(test2(M, S, N, opDeclSet(M), VDS, empty-set)))
        TestS)) .
eq genTestSets(M, none, N, VDS, TestS) = < set(M, VDS) ; TestS > .

ceq test2(M, S, N, ((op F : SL -> S' [AS] .) ODS), VDS, TS)
  = if sortLeq(M, S', S)
    then test2(M, S, p1(test3(N, SL, VDS)), ODS,
           (VDS p2(test3(N, SL, VDS))),
           termSet(TS, F[p3(test3(N, SL, VDS))]))
    else test2(M, S, N, ODS, VDS, TS)
    fi
    if (SL =/= nil) and (ctor in AS) .
ceq test2(M, S, N, ((op F : nil -> S' [AS] .) ODS), VDS, TS)
  = if sortLeq(M, S', S)
    then test2(M, S, N, ODS, VDS, termSet(TS, F))
    else test2(M, S, N, ODS, VDS, TS)
    fi
    if ctor in AS .
ceq test2(M, S, N, ((op F : SL -> S' [AS] .) ODS), VDS, TS)
  = test2(M, S, N, ODS, VDS, TS)
    if not ctor in AS .
eq test2(M, S, N, none, VDS, TS) = (N, VDS, TS) .

eq test3(N, (S SL), ((var X : S .) VDS))
  = test4(N, SL, VDS, X, (var X : S .)) .
ceq test3(N, (S SL), VDS)
  = test4(N + 1, SL, VDS, index(S, N), (var index(S, N) : S .))
    if not isThereVariable?(VDS, S) .
```



```
  eq test4(N, (S SL), ((var X : S .) VDS), TL, VDS')
    = test4(N, SL, VDS, (TL, X), ((var X : S .) VDS')) .
  ceq test4(N, (S SL), VDS, TL, VDS')
    = test4(N + 1, SL, VDS, (TL, index(S, N)),
                                 ((var index(S, N) : S .) VDS'))
      if not isThereVariable?(VDS, S) .
  eq test4(N, nil, VDS, TL, VDS') = < N ; VDS VDS' ; TL > .
 endfm)
```

Let us illustrate the use of the function `genTestSets` by giving a concrete example. Let us consider the following specification of natural numbers with sorts `Nat` of natural numbers and `NzNat` of nonzero natural numbers defined using Peano notation. Notice the use of the attribute `ctor` to indicate the operations used as constructors.

```
(fmod NAT is
   sorts NzNat Nat .
   subsort NzNat < Nat .
   op 0 : -> Nat [ctor] .
   op s_ : Nat -> NzNat [ctor] .
   op _+_ : Nat Nat -> Nat [comm assoc id: 0] .
   vars N M : Nat .
   eq s N + s M = s s (N + M) .
 endfm)
```

The function `genTestSets` takes as parameter the metarepresentation of a module. We can call it using explicitly the metarepresentation of such a module, or we can call it using the `up` function described in Section 10.1.4. For example, we can have the following reduction:

```
Maude> (reduce genTestSets(up(NAT)) .)
result TestSetSolution :
    < fmod 'NAT is
         including 'BOOL .
         sorts 'NzNat ; 'Nat .
         subsort 'NzNat < 'Nat .
         op '0 : nil -> 'Nat [ ctor ] .
         op '_+_ : 'Nat 'Nat -> 'Nat [ assoc comm id ( { '0 } 'Nat ) ] .
         op 's_ : 'Nat -> 'NzNat [ ctor ] .
         var 'M : 'Nat .
         var 'N : 'Nat .
         none
         eq '_+_ [ 's_ [ 'N ] , 's_ [ 'M ] ]
           = 's_ [ 's_ [ '_+_ [ 'N , 'M ] ] ] .
```

```
endfm ;
test ( 'NzNat , 's_ [ 'M ] )
test ( 'Nat , termSet ( '0 , 's_ [ 'M ] ) ) >
```

11 Internal strategies

As explained in Sections 2.2 and 5.1, the operational semantics for functional modules is equational simplification, that is, rewriting of terms until a canonical form is reached. This computational strategy is certainly adequate for the intended use of functional modules as functional programs. A functional module corresponds to a membership equational theory that is Church-Rosser and terminating, and the expected query is the equality between a functional expression and its value.

Maude's default operational semantics for system modules reduces the input terms using the rules as much as possible in a fair top-down fashion. System modules are rewrite theories that need not be Church-Rosser or terminating, so rewriting can go in many undesired directions. Therefore, as opposed to equational simplification for functional modules, this *default strategy* for system modules is inadequate for most intended applications.

In what follows we first argue via examples that no specific operational semantics will be adequate for a sufficiently broad class of system modules and possible queries. Next, we explain the concept of *internal strategies* and how the user can tailor the default strategy for system modules to fit his/her particular computational needs. The idea is that, by using the operations `meta-reduce`, `meta-rewrite`, and `meta-apply` as basic building blocks, the user can easily define computational strategies with equations and rewrite rules. In this way, strategies become internalized within rewriting logic, so that they have a logical semantics and can be reasoned about as other rewrite theories. Finally, we provide a range of simple examples to illustrate the use of the specific metalevel facilities of Maude for defining computational strategies. In particular, we explain the different uses of the operation `meta-apply` for defining basic computational strategies. More complex examples of user-defined internal strategies can be found in [4, 11].

As a source of examples for different computational strategies, consider the following module `YA-BLOCKS-WORLD`. This module specifies a simple blocks world, similar to the ones described in Section 8.4 and 9.5, consisting of six colored blocks (`yellow`, `red`, `green`, `black`, `blue`, and `grey`) and a table. For the sake of the example, we assume that each block has a size, defined by the operation `size`. As usual, a block can be piled on top of another one, and can be moved from the top of a pile to the table and vice-versa. A scenario in the blocks world is represented by giving the set of all its relevant facts. In particular, to represent that a block is on the table we use the unary constructor `onTable`; to represent that a block is piled

right on top of another one we use the binary constructor `onBlock`; and, finally, to represent that a block has no other blocks on top, we use the unary constructor `clear`.

```
mod YA-BLOCKS-WORLD is
  protecting MACHINE-INT .
  sorts Block Fact FactSet .
  subsort Fact < FactSet .

  ops blue green red yellow black grey : -> Block [ctor] .
  op onBlock : Block Block -> Fact [ctor] .
  op onTable : Block -> Fact [ctor] .
  op clear : Block -> Fact [ctor] .
  op factSet : FactSet FactSet -> FactSet [ctor assoc comm] .
  op size : Block -> MachineInt .

  eq size(blue) = 1 .   eq size(green) = 2 .
  eq size(red) =  3 .   eq size(yellow) = 4 .
  eq size(black) = 5 .  eq size(grey) = 6 .

  vars X Y Z : Block .

  *** move a block X from a block Z to the table
  rl [unstack] : factSet(onBlock(X, Z), clear(X))
             => factSet(onTable(X), clear(X), clear(Z)) .

  *** move a block X from the table to a block Z
  rl [stack] : factSet(onTable(X), clear(X), clear(Z))
           => factSet(onBlock(X, Z), clear(X)) .

  *** move a block X from a block Z to another block Y
  rl [move] : factSet(onBlock(X, Z), clear(X), clear(Y))
          => factSet(onBlock(X, Y), clear(X), clear(Z)) .
endm
```

Sets of facts are then represented with the associative and commutative constructor `factSet`. For example, the scenario A depicted in Figure 6 is represented by the following term of sort `FactSet`:

```
factSet(onBlock(blue, green), onBlock(yellow, red),
      onTable(green), onTable(red),
      clear(blue), clear(yellow)) .
```

The initial model described by the module `YA-BLOCKS-WORLD` is the transition system containing exactly all the possible moves allowed in this blocks world, where

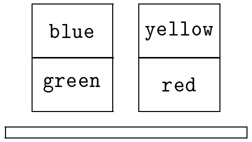


Figure 6: Scenario A.

the rules `unstack`, `stack`, and `move` represent the basic moves. Of course, depending on the application at hand, we need to apply the rules in `YA-BLOCKS-WORLD` in different ways, and the problem is that most of them are not expressible in Maude's default strategy for applying rules in system modules. For example, one application may require piling the `yellow` block on top of any other block; another application may instead require computing whatever transition is needed to reach a particular scenario, for example, the one depicted in Figure 6; and a given problem situation may impose further constraints on the moves that are allowed in reaching that final scenario.

Maude's default strategy for system modules is clearly inadequate for any of these applications: to begin with, the rewrite rules in `YA-BLOCKS-WORLD` are nonterminating. The fact is that, no matter what strategy is implemented, we can always find an application for which it turns out to be inadequate. However, Maude offers a solution for this problem, namely, using the reflective capabilities of Maude to define *inside* the logic whatever computational strategy may be required by the application at hand.

The idea is as follows. A computational strategy S can be thought of as a (partial) computable function over modules and terms which, when it terminates, yields the term resulting from rewriting the initial term with the rules in the module according to the given strategy. Such a strategy can then be equationally defined at the metalevel as a function $\overline{S}$ over terms that metarepresent modules and terms; if this definition is correct, evaluating the functional term $\overline{S}(\overline{M}, \overline{t})$ will correspond to rewriting the term t in the module M with the strategy S. We call the metalevel function $\overline{S}$ an *internal strategy*, since it is defined *inside* the logic using reflection. A more abstract account of internal strategies for declarative programming languages is given in [4].

In the rest of this section, we explain via examples the key facilities available in Maude for defining internal strategies, by means of the built-in operations `meta-apply` and `meta-reduce`. We define these internal strategies in a module called `STRATEGY`.

Consider that the application at hand requires moving an (arbitrary) block from the top of a pile to the table. This is a particular case of a more general computa-

tional strategy that applies a rule once at the top of a given term; in our example, the rule to be applied is the rule `unstack`. Using the built-in operation `meta-apply`, this general computational strategy is easily and efficiently defined at the metalevel as the operation `apply` below.

```
fmod STRATEGY is
  protecting META-LEVEL .
  op apply : Module Term Qid -> Term .
  op extTerm : ResultPair -> Term .
  op extSubst : ResultPair -> Substitution .

  var M : Module .   var T : Term .
  var L : Qid .      var SB : Substitution .

  eq apply(M, T, L) = extTerm(meta-apply(M, T, L, none, 0)) .

  eq extTerm({T, SB}) = T .
  eq extSubst({T, SB}) = SB .
```

The operations `extTerm` and `extSubst` are selectors extracting the first and second component, respectively, from a pair constructed with `{_,_}`.

Thus, evaluating the functional term

```
  apply(YA-BLOCKS-WORLD,
        factSet(onBlock(blue, green), onBlock(yellow, red),
                onTable(green), onTable(red),
                clear(blue), clear(yellow)),
        unstack)
```

corresponds to computing the transformation required by the application we are considering, when the initial scenario is the one depicted in Figure 6.

As explained in Section 10.1.3, the built-in operation `meta-apply` can also represent a single application of a rule partially instantiated with a substitution. This facility is of course needed when applying rules whose righthand sides or conditions contain variables that do not appear in their lefthand sides; but it is also key when defining computational strategies that impose constraints on the application of rules; our next example illustrates this particular use of `meta-apply`.

Consider now that the application at hand requires moving the `yellow` block from the top of a pile to the top of a different pile. This is a particular case of a more general computational strategy that applies a rule partially instantiated with a substitution; in our example, the rule to be applied is the rule `move`, with the variable `X` substituted by the term `yellow`. Using the built-in operation `meta-apply`, this general computational strategy is easily and efficiently defined at the metalevel as the operation `applyWithSubst` below.

```
op applyWithSubst : Module Term Qid Substitution -> Term .

eq applyWithSubst(M, T, L, SB) = extTerm(meta-apply(M, T, L, SB, 0)) .
```

In particular, evaluating the functional term

```
applyWithSubst(YA-BLOCKS-WORLD,
               factSet(onBlock(blue, green), onBlock(yellow, red),
                       onTable(green), onTable(red),
                       clear(blue), clear(yellow)),
               move, (X <- yellow))
```

corresponds to computing the transformation required by the application we are considering, when the initial scenario is the one depicted in Figure 6.

However, many applications require not only that in the substitution with which a rule is applied certain values are assigned to certain variables, but also that those values satisfy certain properties. In these cases, a proper combination of the operations `meta-apply` and `meta-reduce`, along with the capability of moving efficiently between levels in the reflective tower, is very useful to define the corresponding computational strategies.

Consider the case when the satisfaction of the condition restricting the application of a rule only depends on the substitution with which the rule is to be applied (see [4] for more complex examples). This condition can then be represented as a Boolean expression over the variables that appear in the rule.

Remember that the operation `meta-apply` can be used to represent the application of a rule at the top of a term with any of its successful matches. The operation **`applyIf`** below uses this capability to define at the metalevel the general computational strategy we are considering in this example. The fourth argument of this operation is the metarepresentation of the condition to be satisfied. Note then how we use the operation `meta-reduce` to find among all the successful matches one that satisfies the required condition. Finally, the operation `substitute` performs at the metalevel the operation of substitution.

```
op applyIf : Module Term Qid Term MachineInt -> Term .
op substitute : Term Substitution -> Term .
op substituteAux : TermList Substitution -> TermList .

var C : Term .

eq applyIf(M, T, L, C, N)
  = if meta-apply(M, T, L, none, N) == {error*, none}
    then error*
    else (if meta-reduce(M, substitute(C,
```



```
                                   extSubst(meta-apply(M, T, L, none, N))))
              == {'true}'Bool
           then extTerm(meta-apply(M, T, L, none, N))
           else  applyIf(M, T, L, C, (N + 1)) fi) fi .

vars F S X Y : Qid .   var TL : TermList .

eq substitute(T, none) = T .
eq substitute(X, ((Y <- T); SB))
  = if X == Y then T else substitute(X, SB) fi .
eq substitute({F}S, SB) = {F}S .
eq substitute(F[TL], SB) = F[substituteAux(TL, SB)] .
eq substituteAux((T, TL), SB)
  = (substitute(T, SB), substituteAux(TL, SB)) .
eq substituteAux(T, SB) = substitute(T, SB) .
```

To illustrate the use of `applyIf`, consider an application that requires moving a block x from a block z to another block y, if y has smaller size than x. Thus, evaluating the functional term

```
applyIf(YA-BLOCKS-WORLD,
        factSet(onBlock(blue, green), onBlock(yellow, red),
                onTable(green), onTable(red),
                clear(blue), clear(yellow)),
        move, (size(Y) < size(X)), 0)
```

corresponds to computing the transformation required by the application we are considering, when the initial scenario is the one depicted in Figure 6.

Notice that by using the operation `applyIf`, we can turn, if needed, an unconditional rule into a conditional one, without changing the original module. This is just a simple example of the modularity that results from separating specification and control via reflection.

Finally, consider an application that requires finding a sequence of moves that transforms an arbitrary scenario into a final one; its computational strategy corresponds to a search algorithm. This application is an instance of a more general class of applications that require finding a sequence of rewrites that reduce an arbitrary term into a final one. Games and planners are applications that belong to this class; the rules of the game or the actions to be planned are represented as rewrite rules, and the winning state or the final state are represented as terms.

In what follows we define the computational strategy that corresponds to a breadth-first search; other search algorithms can be similarly defined. The idea is to use the operation `meta-apply` to generate from the initial term the corresponding tree of possible rewriting sequences, and to use the operation `meta-reduce` to check whether any of the rewriting sequences has reached the final term; the nodes of the state tree represent intermediate steps in the possible rewriting sequences.

First, we define the data type `Tree` for representing state trees. The idea is that the root node of a state tree represents the initial term; each branch in the tree represents a possible rewriting sequence starting from the initial term; and each node in a branch represents an intermediate step in the corresponding rewriting sequence; a solution is then the list of nodes that form a branch leading to the final term. In particular, a state tree is built with the constructor `tree` that takes two arguments: a node and a list of trees. A node is a pair built with the constructor `step`; its first element is the rule that is applied at this intermediate step of the rewriting sequence, and its second element is the pair formed by the term that results from applying this rule and the substitution that is used in its application. Each node in a tree occupies a position that is represented as a list of natural numbers with the `cons`-like constructor `pos`: the position `nilPos` corresponds to the root node, and the position `pos`(n, p) corresponds to the node that occupies the position p in the nth-child of the root node. A list of trees is built with the `cons`-like constructor `treeL`; the empty list of trees is represented with the constant `nilTreeL`. Similarly, a list of nodes is built with the `cons`-like constructor `sol`; the empty list of nodes is represented with the constant `nilSol`.

```
sorts Node Tree TreeList Solution Position .

op step : Qid ResultPair -> Node [ctor] .
op tree : Node TreeList -> Tree [ctor] .
op treeL : Tree TreeList -> TreeList [ctor] .
op nilTreeL : -> TreeList [ctor] .

op pos : MachineInt Position -> Position [ctor] .
op nilPos : -> Position [ctor] .

op sol : Node Solution -> Solution [ctor] .
op nilSol : -> Solution [ctor] .
```

Next, we declare a number of auxiliary operations over state trees:

- `nextPos`(tr, p) returns the position occupied by the next node to be expanded in a breadth-first search, when the current node occupies the position p in a tree tr.
- `getNode`(tr) extracts the root node from a tree tr.
- `extTermNode`(nd) extracts the first element from a node nd.

- `getSubtree`(tr, p) extracts from a tree tr the subtree whose root node occupies the position p.
- `addSubTree`(tr, tr', p) extends the tree tr with a new subtree tr' so that tr' is a new child of the node that occupies the position p.
- `extSol`(tr, p) forms the list of all the nodes in a tree tr that appear in the branch leading to the node that occupies the position p.
- `appendSol`(sl, sl') appends two lists of nodes.

```
op nextPos : Tree Position -> Position .
op right : Position -> Position .
op down : Position -> Position .
op getNode? : Tree  -> Bool .
op getNode : Tree  -> Node .
op extTermNode : Node -> Term .
op getSubTree : Tree Position -> Tree .
op getSubTreeL : TreeList MachineInt -> Tree .
op addSubTree : Tree Tree Position -> Tree .
op addSubTreeL : TreeList Tree Position -> TreeList .
op appendTreeL : TreeList TreeList -> TreeList .
op extSol : Tree Position -> Solution .
op extSolAux : TreeList Position -> Solution .
op appendSol : Solution Solution -> Solution .

vars QL QL' : QidList .  var Nd : Node .
vars Tr Tr' : Tree .     vars TrL TrL' : TreeList .
var P : Position .       vars Sol Sol' : Solution .
var N : MachineInt .

eq nextPos(Tr, P)
  = if getNode?(getSubTree(Tr, right(P))) == true
    then right(P) else down(P) fi .

eq right(pos(N, P))
  = if P == nilPos then pos((N + 1), P) else pos(N, right(P)) fi .

eq down(nilPos) = pos(1, nilPos) .
eq down(pos(N, P)) = pos(N, down(P)) .

eq getNode?(tree(Nd, TrL)) = true .

eq getNode(tree(Nd, TrL)) = Nd .
```

```
  eq extTermNode(step(L, {T, SB})) = T .

  eq getSubTree(Tr, nilPos) = Tr .

  eq getSubTree(tree(Nd, TrL), pos(N, P))
    = getSubTree(getSubTreeL(TrL, N), P) .

  eq getSubTreeL(treeL(Tr, TrL), N)
    = if N == 1 then Tr else getSubTreeL(TrL, _-_(N, 1)) fi .

  eq addSubTree(tree(Nd, TrL), Tr, nilPos)
    = tree(Nd, appendTreeL(TrL, treeL(Tr, nilTreeL))) .

  eq addSubTree(tree(Nd, TrL), Tr, pos(N, P))
    = tree(Nd, addSubTreeL(TrL, Tr, pos(N, P))) .

  eq addSubTreeL(treeL(Tr, TrL), Tr', pos(N, P))
    = if N == 1
      then treeL(addSubTree(Tr, Tr', P), TrL)
      else treeL(Tr, addSubTreeL(TrL, Tr', pos(_-_(N, 1), P))) fi .

  eq appendTreeL(nilTreeL, TrL) = TrL .
  eq appendTreeL(treeL(Tr, TrL), TrL')
    = treeL(Tr, appendTreeL(TrL, TrL')) .

  eq extSol(tree(Nd, TrL), nilPos) = sol(Nd, nilSol) .
  eq extSol(tree(Nd, TrL), pos(N, P))
    = sol(Nd, extSolAux(TrL, pos(N, P))) .

  eq extSolAux(treeL(Tr, TrL), pos(N, P))
    = if N == 1 then extSol(Tr, P)
                else extSolAux(TrL, pos(_-_(N, 1), P)) fi .

  eq appendSol(nilSol, Sol) = Sol .
  eq appendSol(sol(Nd, Sol), Sol') = sol(Nd, appendSol(Sol, Sol')) .
```

Now we introduce the operation `findPlan` below, that defines the computational strategy for finding a rewriting sequence leading to a final term, when the search follows a breadth-first strategy. In particular, $\texttt{findPlan}(M, tr, p, t, ql, ql', n)$ defines for a system module M the computational strategy for finding the rewriting sequence that leads to a final term t, following a breadth-first search and using only the rules in the list ql'; the second argument tr of this operation represents the state tree; its third argument p is the position of the node from which the search is to be

continued; its fourth argument ql is the list of rules still to be applied to that node; and its last argument n is the number of the match that will be considered next when attempting to extend the search with the first rule in the list ql. Then, for each node in the state tree, the operation `findPlan` uses the operation `meta-apply` to extend the tree below that node with the list of trees whose root nodes represent a possible next step in the rewriting sequence. Before adding a new node to the state tree, the operation `findPlanAux` is used to check whether that node corresponds to the final term; if this is the case, the operation `findPlanAux` returns the list of nodes in the branch that leads to that node.

Note that for any scenario with at least two blocks there is always a rule in `YA-BLOCKS-WORLD` that can be applied. Thus, if the final scenario is reachable from the initial one, the operation `findPlan` will find the rewriting secuence that leads from one to the other; otherwise, the operation `findPlan` will not terminate.

```
  op findPlan : Module Tree Position Term
                QidList QidList MachineInt -> Solution .

  op findPlanAux : Module Tree Position Term
                   QidList QidList MachineInt Node -> Solution .

  var RP : ResultPair .

  eq findPlan(M, Tr, P, T, (L QL), QL', N)
    = if meta-apply(M, extTermNode(getNode(getSubTree(Tr, P))),
                    L, none, N)  == {error*, none}
      then findPlan(M, Tr, P, T, QL, QL', 0)
      else findPlanAux(M, Tr, P, T, (L QL), QL', N,
          step(L, meta-apply(M, extTermNode(getNode(getSubTree(Tr, P))),
                             L, none, N)))
      fi .

  eq findPlan(M, Tr, P, T, nil, QL', N)
    = findPlan(M, Tr, nextPos(Tr, P), T, QL', QL', 0) .

  eq findPlanAux(M, Tr, P, T, QL, QL', N, step(L, RP))
    = if meta-reduce(M, '_==_[T, extTerm(RP)]) == {'true}'Bool
      then appendSol(extSol(Tr, P), sol(step(L, RP), nilSol))
      else findPlan(M, addSubTree(Tr, tree(step(L, RP), nilTreeL), P),
                    P, T, QL, QL', (N + 1))
      fi .
endfm
```

To illustrate the use of the operation `findPlan`, consider an application that

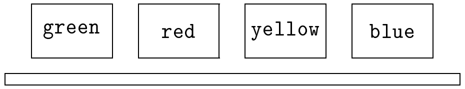


Figure 7: Scenario B.

requires finding a sequence of moves that transforms the scenario B depicted in Figure 7 into the scenario A showed in Figure 6. The evaluation of the term

```
findPlan(YA-BLOCKS-WORLD,
         tree(step('initial,
                   {factSet(onTable(blue), onTable(green),
                            onTable(yellow), onTable(red),
                            clear(blue), clear(green),
                            clear(yellow), clear(red)),
                    none}),
              nilTreeL),
         nilPos,
         factSet(onBlock(blue, green), onBlock(yellow, red),
                 onTable(green), onTable(red),
                 clear(blue), clear(yellow)),
         (unstack stack move), (unstack stack move), 0)
```

corresponds to implementing and executing the corresponding planner.

See the case studies in the Maude web page at `http://maude.csl.sri.com` for several examples of search strategies applied to the analysis of communication protocols (see also [11]). Of course, as is well-known, breadth-first search can, in a *direct* implementation such as the one described above, be very space inefficient, due to the combinatorial explosion resulting from the search. A good practical alternative used in the case studies just mentioned is simulating breadth-first search by depth-first search with iterated depth. Our concern here is not efficiency, but rather illustrating the general way in which strategies can be defined and used.

Acknowledgements

We thank all our collaborators along the years for all their contributions to the system, and all our colleagues working on similar systems such as CafeOBJ and ELAN for interesting discussions and valuable suggestions. We are very grateful to Alberto Verdejo and Miguel Palomino for their detailed comments and corrections to previous versions of this document, and to Olga Marroquín Alonso for her help in the Petri net model of a library in Section 8.3.

The work reported here has been supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by DARPA and NASA through Contract NAS2-98073, by Office of Naval Research Contract N00014-99-C-0198, and by National Science Foundation Grant CCR-9900334.

References

[1] E. Astesiano, H.-J. Kreowski, and B. Krieg-Bruckner, editors. *Algebraic Foundations of Systems Specification*. Springer, 1998.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. To appear in *Theoretical Computer Science*. Short version in M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, LNCS 1214, pages 67–92. Springer, 1997.

[4] M. Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language*. Ph.D. thesis, University of Navarre, Spain, February 1998. To be published by CSLI Publications, Stanford University.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In C. Kirchner and H. Kirchner, editors, *Proc. Second Int. Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, France, September 1998*, Electronic Notes in Theoretical Computer Science 15. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and programming in rewriting logic*. Technical Report, Computer Science Laboratory, SRI International, January 1999, revised August 1999. `http://maude.csl.sri.com`

[7] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Design and implementation of the Cafe prover and the Church-Rosser checker tools. Technical Report, Computer Science Laboratory, SRI International, February 1998.

[8] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational logic tools by reflection in rewriting logic. In *Proc. CafeOBJ Symposium '98, Numazu, Japan*, CafeOBJ Project, April 1998.

[9] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing and J. Woodcock, editors, *FM'99 — Formal Methods*, LNCS 1709, pages 1684–1703. Springer, 1999.

[10] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. First Int. Workshop on Rewriting Logic and its Applications, Asilomar, California, September 1996*, Electronic Notes in Theoretical Computer Science 4. Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`

[11] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998. `http://www.cs.bell-labs.com/who/nch/fmsp/index.html`

[12] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen *et al.*, editors, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pages 243–320. The MIT Press/Elsevier, 1990.

[13] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. Ph.D. thesis, University of Málaga, Spain, June 1999.

[14] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985.

[15] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, 1987.

[16] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions, and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[17] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley and Sons, 1990.

[18] C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ-3. In T. Lepistö and A. Salomaa, editors, *Proc. ICALP'88*, LNCS 317, pages 287–301. Springer, 1988.

[19] J. Loeckx, H.-D. Ehrich, and M. Wolf, *Specification of Abstract Data Types*. Wiley Teubner, 1996.

[20] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. To appear in D. M. Gabbay, editor, *Handbook of Philosophical Logic, Second edition, Volume 6*. Kluwer Academic Publishers. Short version in J. Meseguer, editor, *Proc. First Int. Workshop on Rewriting Logic and its Applications, Asilomar, California, September 1996*, Electronic Notes in Theoretical Computer Science 4. Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`

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

[22] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

[23] J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *Proc. CONCUR'96, Pisa, August 1996*, LNCS 1119, pages 331–372. Springer, 1996.

[24] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th Int. Workshop, WADT'97, Tarquinia, Italy, June 1997*, LNCS 1376, pages 18–61. Springer, 1998.

[25] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, NATO ASI Series F: Computer and Systems Sciences 165, pages 347–398. Springer, 1998.

[26] J. Meseguer and J. A. Goguen. Initiality, induction, and computability. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.

[27] W. Reisig. *Petri Nets: An Introduction*. Springer, 1985.

[28] J. Siekmann and P. Szabó. A Noetherian and confluent rewrite system for idempotent semigroups. *Semigroup Forum*, 25:83–110, 1982.

[29] A. Verdejo and N. Martí-Oliet. *Executing and verifying CCS in Maude*. Technical Report 99–00, Depto. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain, February 2000.

[30] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, LNCS 817, pages 648–660. Springer, 1994.