Maude Manual
(Version 3.1)

Manuel Clavel
Francisco Durán
Steven Eker
Santiago Escobar
Patrick Lincoln
Narciso Martí-Oliet
José Meseguer
Rubén Rubio
Carolyn Talcott

October 2020

Contents

List of Figures

Chapter 1

Introduction

This introduction tries to give the big picture on the goals, design philosophy, logical foundations, applications, and overall structure of Maude. It is written in an impressionistic, conversational style, and should be read in that spirit. The fact that occasionally some particular technical concept mentioned in passing (for example, "the Church-Rosser property") may be unfamiliar should not be seen as an obstacle. It should be taken in a relaxed, sporting spirit: those things will become clearer in the body of the manual; here it is just a matter of gaining a first overall impression.

1.1 Simplicity, expressiveness, and performance

Maude's language design can be understood as an effort to simultaneously maximize three dimensions:

- *Simplicity*: programs should be as simple as possible and have clear meaning.
- *Expressiveness*: a very wide range of applications should be naturally expressible: from sequential, deterministic systems to highly concurrent nondeterministic ones; from small applications to large systems; and from concrete implementations to abstract specifications, all the way to *logical frameworks*, in which not just applications, but entire formalisms, other languages, and other logics can be naturally expressed.
- *Performance*: concrete implementations should yield system performance competive with other efficient programming languages.

Although simplicity and performance are natural allies, maximizing expressiveness is perhaps the key point in Maude's language design. Languages are after all *representational devices*, and their merits should be judged on the degree to which problems and applications can be represented and reasoned about generally, naturally, and easily. Of course, *domain-specific* languages also have an important role to play in certain application areas, and can offer a useful "economy of representation" for a given area. In this regard, Maude should be viewed as a high-performance *metalanguage*, through which many different domain-specific languages can be developed.

1.1.1 Simplicity

Maude's basic programming statements are very simple and easy to understand. They are *equations* and *rules*, and have in both cases a simple *rewriting semantics* in which instances of

the lefthand side pattern are replaced by corresponding instances of the righthand side.

A Maude program containing only equations is called a *functional module*. It is a functional program defining one or more functions by means of equations, used as simplification rules. For example, if we build lists of quoted identifiers (which are sequences of characters starting with the character '’' and belong to a sort[1] `Qid`) with a "cons" operator denoted by an infix period,

```
op nil : -> List .
op _._ : Qid List -> List .
```

then we can define a length function and a membership predicate by means of the operators and equations

```
op length : List -> Nat .
op _in_ : Qid List -> Bool .

vars I J : Qid .
var  L : List .

eq length(nil) = 0 .
eq length(I . L) = s length(L) .

eq I in nil = false .
eq I in J . L = (I == J) or (I in L) .
```

where `s_` denotes the successor function on natural numbers, `_==_` is the equality predicate on quoted identifiers, and `_or_` is the usual disjunction on Boolean values. Such equations (specified in Maude with the keyword `eq` and ended with a period) are used from left to right as *equational simplification rules*. For example, if we want to evaluate the expression

```
length('a . 'b . 'c . nil)
```

we can apply the second equation for `length` to simplify the expression three times, and then apply the first equation once to get the final value `s s s 0`:

```
length('a . 'b . 'c . nil)
= s length('b . 'c . nil)
= s s length('c . nil)
= s s s length(nil)
= s s s 0
```

This is the standard "replacement of equals by equals" use of equations in elementary algebra and has a very clear and simple semantics in equational logic. Replacement of equals by equals is here performed only from left to right and is then called *equational simplification* or, alternatively, *equational rewriting*. Of course, the equations in our program should have good properties as "simplification rules" in the sense that their final result exists and should be unique. This is indeed the case for the two functional definitions given above.

In Maude, equations can be *conditional*; that is, they may only be applied if a certain condition holds. For example, we can simplify a fraction to its irreducible form using the conditional equation

```
vars I J : NzInt .
ceq J / I = quot(J, gcd(J, I)) / quot(I, gcd(J, I))
  if gcd(J, I) > s 0 .
```

[1] In Maude, types come in two flavors, called *sorts* and *kinds* (see Section 3, and the discussion of user-definable data in Section 1.1.2 below).

where `ceq` is the Maude keyword introducing conditional equations, `NzInt` is the sort of nonzero integers, and where we assume that the integer quotient (`quot`) and greatest common divisor (`gcd`) operations have already been defined by their corresponding equations.

A Maude program containing rules and possibly equations is called a *system module*. Rules are also computed by rewriting from left to right, that is, as *rewrite rules*, but they are *not* equations; instead, they are understood as *local transition rules* in a possibly concurrent system. Consider, for example, a distributed banking system in which we envision the account objects as floating in a "soup," that is, in a multiset or bag of objects and messages. Such objects and messages can "dance together" in the distributed soup and can interact locally with each other according to specific rewrite rules. We can represent a bank account as a record-like structure with the name of the object, its class name (`Account`) and a `bal`(ance) attribute, say, a natural number. The following are two different account objects in our notation:

```
< 'A-001 : Account | bal : 200 >
< 'A-002 : Account | bal : 150 >
```

Accounts can be updated by receiving different *messages* and changing their state accordingly. For example, we can have `debit` and `credit` messages, such as

```
credit('A-002, 50)
debit('A-001, 25)
```

We can think of the "soup" as formed just by "juxtaposition" (with empty syntax) of objects and messages. For example, the above two objects and two messages form the soup

```
< 'A-001 : Account | bal : 200 >
< 'A-002 : Account | bal : 150 >
credit('A-002, 50)
debit('A-001, 25)
```

in which the order of objects and messages is immaterial. The local interaction rules for crediting and debiting accounts are then expressed in Maude by the rewrite rules

```
var  I : Qid .
vars N M : Nat .

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

crl < I : Account | bal : M > debit(I, N)
  => < I : Account | bal : (M - N) >
  if M >= N .
```

where rules are introduced with the keyword `rl` and conditional rules (like the above rule for `debit` that requires the account to have enough funds) with the `crl` keyword.

Note that these rules *are not equations at all*: they are local transition rules of the distributed banking system. They can be applied concurrently to different fragments of the soup. For example, applying both rules to the soup above we get the new distributed state:

```
< 'A-001 : Account | bal : 175 >
< 'A-002 : Account | bal : 200 >
```

Note that the rewriting performed is *multiset rewriting*, so that, regardless of where the account objects and the messages are placed in the soup, they can always come together and rewrite if a rule applies. In Maude this is specified in the *equational part* of the program (system module) by declaring that the (empty syntax) multiset union operator satisfies the associativity and commutativity equations:

```
X (Y Z) = (X Y) Z
X Y = Y X
```

This is not done by giving the above equations explicitly. It is instead done by declaring the multiset union operator with the `assoc` and `comm` equational attributes (see Section 4.4.1 and Section 1.1.2 below), as follows, where `Configuration` denotes the multisets or soups of objects and messages.

```
op __ : Configuration Configuration -> Configuration [assoc comm] .
```

Maude then uses this information to generate a *multiset matching algorithm*, in which the multiset union operator is matched *modulo* associativity and commutativity.

Again, a program involving such rewrite rules is intuitively very simple, and has a very simple rewriting semantics. Of course, the systems specified by such rules can be highly concurrent and *nondeterministic*; that is, unlike for equations, there is no assumption that all rewrite sequences will lead to the same outcome. For example, depending on the order in which `debit` or `credit` messages are consumed, a bank account can end up in quite different states, because the rule for debiting can only be applied if the account balance is big enough. Furthermore, for some systems there may *not* be any final states: their whole point may be to continuously engage in interactions with their environment as *reactive* systems.

1.1.2 Expressiveness

The above examples illustrate a general fact, namely, that Maude can express with equal ease and naturalness *deterministic* computations, which lead to a unique final result, and *concurrent, nondeterministic* computations. The first kind is typically programmed with equations in functional modules, and the second with rules (and perhaps with some equations for the "data" part) in system modules.

In fact, functional modules define a *functional sublanguage*[2] of Maude. In a functional language true to its name, functions have unique values as their results, and it is neither easy nor natural to deal with highly concurrent and nondeterministic systems while keeping the language's functional semantics. It is well known that such systems pose a serious expressiveness challenge for functional languages. In Maude this challenge is met by system modules, which extend the purely functional semantics of equations to the concurrent rewriting semantics of rules.[3] Although certainly declarative in the sense of having a clear logical semantics, system modules are of course *not* functional: that is their entire *raison d'être*.

Besides this generality in expressing both deterministic and nondeterministic computations, further expressiveness is gained by the following features:

- equational pattern matching,
- user-definable syntax and data,
- types, subtypes, and partiality,
- generic types and modules,
- support for objects, and

[2] This sublanguage is essentially an extension of the OBJ3 equational language [79], which has greatly influenced the design of Maude.

[3] As explained in Section 1.2, mathematically this is achieved by a *logic inclusion*, in which the functional world of equational theories is conservatively embedded in the nonfunctional, concurrent world of rewrite theories.

- reflection.

We briefly discuss each of these features in what follows.

Equational pattern matching

Rewriting with both equations and rules takes place by *matching* a lefthand side term against the subject term to be rewritten. The most common form of matching is *syntactic matching*, in which the lefthand side term is matched as a tree on the (tree representation of the) subject term (see Section 4.7). For example, the matching of the lefthand sides for the equations defining the `length` and `_in_` functions above is performed by syntactic matching. But we have already encountered another, more expressive, form of matching, namely, *equational matching* in the bank accounts example: the lefthand side

```
< I : Account | bal : M > credit(I, N)
```

has the (empty syntax) multiset union operator `__` as its top operator, but, thanks to its `assoc` and `comm` equational attributes, it is matched not as a tree, but as a multiset. Therefore, the match will succeed provided that the subject multiset contains instances of the terms `< I : Account | bal : M >` and `credit(I, N)` in which the variable `I` is instantiated the same way in both terms, *regardless of where those instances appear in the multiset*, that is, *modulo* associativity and commutativity.

In general, a binary operator declared in a Maude module can be defined with any[4] combination of equational attributes of: associativity, commutativity, left-, right-, or two-sided identity, and idempotency. Maude then generates an *equational matching algorithm* for the equational attributes of the different operators in the module, so that each operator is matched *modulo* its equational attributes. This manual will illustrate with various examples the expressive power afforded by this form of equational matching (see Section 4.8).

User-definable syntax and data

In Maude the user can specify each operator with its own syntax, which can be prefix, postfix, infix, or any "mixfix" combination. This is done by indicating with underscores the places where the arguments appear in the mixfix syntax. For example, the infix list *cons* operator above is specified by `_._`, the (empty syntax) multiset union operator by `__`, and the if-then-else operator by `if_then_else_fi`. In practice, this improves readability (and therefore understandability) of programs and data. In particular, for *metalanguage uses*, in which another language or logic is represented in Maude, this can make a big difference for understanding large examples, since the Maude representation can keep essentially the original syntax. The combination of user-definable syntax with equations and equational attributes for matching leads to a very expressive capability for specifying any *user-definable data*. It is well known that any computable data type can be equationally specified [10]. Maude gives users full support for this equational style of defining data which is not restricted to syntactic terms (trees) but can also include lists (modulo associativity), multisets (modulo associativity and commutativity), sets (adding an idempotency equation), and other combinations of equational attributes that can then be used in matching. This great expressiveness for defining data is further enhanced by Maude's rich type structure, as explained below.

Types, subtypes, and partiality

Maude has two varieties of types: *sorts*, which correspond to well-defined data, and *kinds*, which may contain error elements. Sorts can be structured in *subsort hierarchies*, with the subsort relation understood semantically as subset inclusion. For example, for numbers we can have subsort inclusions

[4] Except for any combination including both associativity and idempotency, which is not currently supported.

```
Nat < Int < Rat
```

indicating that the natural numbers are contained in the integers, and these in turn are contained in the rational numbers. All these sorts determine a *kind* (say the "number kind") which is interpreted semantically as the set containing all the well-formed numerical expressions for the above number systems as well as error expressions such as, for example, `4 + 7/0`. This allows support for *partial functions* in a total setting, in the sense that a function whose application to some arguments has a kind but not a sort should be considered *undefined* for those arguments (but notice that functions can also map undefined to defined results, for example in the context of error recovery). Furthermore, operators can be *subsort-overloaded*, providing a useful form of *subtype polymorphism*. For example, the addition operation `_+_` is subsort overloaded and has typings for each of the above number sorts. A further feature, greatly extending the expressive power for specifying partial functions, is the possibility of *defining sorts by means of equational conditions*. For example, a sequential composition operation `_;_` concatenating two paths in a graph is defined if and only if the target of the first path coincides with the source of the second path. In Maude this can be easily expressed with the "conditional membership" (see Section 4.3):

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

Generic types and modules

Maude supports a powerful form of *generic programming* that substantially extends the *parameterized programming* capabilities of OBJ3 [79]. The analogous terminology to express these capabilities in higher-order type theory would be *parametric polymorphism* and *dependent types*. But in Maude the parameters are not just types, but *theories*, including operators and equations that impose semantic restrictions on the parameterized module instantiations. Thus, whereas a parametric `LIST` module can be understood just at the level of the parametric type (sort) of list elements, a parameterized `SORTING` module has the theory `TOSET` of totally ordered sets as its parameter, including the axioms for the order predicate, that must be satisfied in each correct instance for the sorting function to work properly. Types analogous to dependent types are also supported by making the parameter instantiations depend on specific parametric constants in the parameter theory, and by giving membership axioms depending on such constants. For example, natural numbers modulo n (see Section 22.7), and arrays of length n, can be easily defined this way. The fact that entire modules, and not just types, can be parametric provides even more powerful constructs. For example, `TUPLE[`n`]` (see Section 21.3.1) is a "dependent parameterized module" that assigns to each natural number n the parameterized module of n-tuples (together with the tupling and projection operations) with n parameter sorts.

Support for objects

The bank accounts example illustrates a general point, namely, that in Maude it is very easy to support objects and distributed object interactions in a completely declarative style with rewrite rules. Although such object systems are just a particular style of system modules in which object interactions (through messages or directly between objects) are expressed by rewriting, Maude provides special support for object-based programming and for fair execution of object-based applications (see Chapter 8). Furthermore, the Full Maude extension provides special syntax in *object-oriented modules* (see Chapter 22). Such modules directly support object-oriented concepts like objects, messages, classes, and multiple class inheritance. Moreover, the support for *communication with external objects* (see Section 9) allows Maude objects to interact by message passing with internet sockets and, through them, with all kinds of other external

objects, such as files, databases, graphical user interfaces, sensors, robots, and so on. All this is achieved without compromising Maude's declarative nature: interaction with normal Maude objects and with external objects can both be programmed with rewrite rules. Using internet sockets as external objects, it is also easy to develop *distributed implementations* in Maude, where a "soup" of objects and messages is not realized just as a multiset data structure in a single sequential machine, but as a "distributed soup," with objects and messages in different machines or in transit.

Reflection

This is a very important feature of Maude. Intuitively, it means that Maude programs can be metarepresented as *data*, which can then be manipulated and transformed by appropriate functions. It also means that there is a systematic *causal connection* between Maude modules themselves and their metarepresentations, in the sense that we can either first perform a computation in a module and then metarepresent its result, or, equivalently, we can first metarepresent the module and its initial state and then perform the entire computation *at the metalevel*. Finally, the metarepresentation process can itself be iterated giving rise to a very useful *reflective tower*. Thanks to Maude's logical semantics (more on this in Section 1.2), all this is not just some kind of "glorified hacking," but a precise form of *logical reflection* with a well-defined semantics (see Chapter 17 and [33, 34]). There are many important applications of reflection. Let us mention just three:

- *Internal strategies*. Since the rewrite rules of a system module can be highly nondeterministic, there may be many possible ways in which they can be applied, leading to quite different outcomes. In a distributed object system this may be just part of life: provided some fairness assumptions are respected, any concurrent execution may be acceptable. But what should be done in a sequential execution? Maude does indeed support two different fair execution strategies in a built-in way through its `rewrite` and `frewrite` commands (see Section 5.4). But what if we want to use a different strategy for a given application? The answer is that Maude modules can be executed at the metalevel with user-definable *internal strategies*[5] (see Section 17.7). Such internal strategies can be defined by rewrite rules in a metalevel module that guides the possibly nondeterministic application of the rules in the given "object level" module. This process can be iterated in the reflective tower. That is, we can define meta-strategies, meta-meta-strategies, and so on.

- *Module algebra*. The entire *module algebra* in which parameterized modules can be composed and instantiated becomes expressible within the logic, and *extensible* by new module operations that transform existing modules metarepresented as data. This is of more than theoretical interest: Maude's module algebra is realized exactly in this way by Full Maude, a Maude program defining all the module operations and easily extensible with new ones (see Part II of this manual).

- *Formal tools*. The verification tools in Maude's formal environment must take Maude modules as arguments and perform different formal analyses and transformations on such modules. This is again done by reflection in tools such as Maude's inductive theorem prover, the Church-Rosser checker, the Maude termination tool, the Real-Time Maude tool, and so on.

[5] That is, internal to Maude's logic, in the sense of being definable by logical axioms.

1.1.3 Performance

Achieving expressiveness in all the ways described above without sacrificing performance is a nontrivial matter. Successive Maude implementations have been advancing this goal while expanding the set of language features. More work remains ahead, but it seems fair to say that Maude, although still an interpreter, is a high-performance system that can be used for many non-toy applications with competitive performance and with many advantages over conventional code. Without in any way trying to extrapolate a specific experience into a general conclusion, a concrete example from the Maude user's trenches may illustrate the point. A formal tool component to check whether a trace of events satisfies a given linear temporal logic (LTL) formula was written in Maude at NASA Ames by Grigore Roşu in about one page of Maude code. The component had a trivial correctness proof—the Maude module was based on the equational definition of the LTL semantics for the different connectives. This replaced a similar component having about 5,000 lines of Java code that had taken over a month to develop by an experienced colleague. The Java tool used a translation of LTL formulas into Büchi automata (the usual method to efficiently model check an LTL formula) and run about three times more slowly than the Maude code. It would have been very difficult to prove the correctness of the Java tool and, having a better and clearly correct alternative in the Maude implementation, this was never done.

Generally and roughly speaking, the current Maude implementation can execute syntactic rewriting with typical speeds from half a million to several million rewrites per second, depending on the particular application and the given machine. Similarly, associative and associative-commutative equational rewriting with term patterns used in practice[6] can be performed at the typical rate of one hundred thousand to several hundred thousand rewrites per second.

These figures must be qualified by the observation that, until recently, the cost of an associative or associative-commutative rewriting step *depended polynomially on the size of the subject term*, even with the most efficient algorithms. In practice this meant that this kind of rewriting was not practical for large applications, in which the lists or multisets to be rewritten could have millions of elements. This situation has been drastically altered by a recent result of Steven Eker [59] providing new algorithms for associative and associative-commutative rewriting that, for patterns typically encountered in practice, can perform one step of associative rewriting in constant time, and one associative-commutative rewriting step in time proportional to the logarithm of the subject term's size. Maude supports equational rewriting with these new algorithms.

The reason why the Maude interpreter achieves high performance is that the rewrite rules are carefully analyzed and are then *semicompiled* into efficient matching and replacement automata [57] with efficient matching algorithms. One important advantage of semicompilation is that it is possible to trace every single rewriting step. More performance is of course possible by full compilation. Maude has an experimental compiler for a subset of the language which can typically achieve a fivefold speedup over the interpreter.

Four other language features give the user different ways of optimizing the performance of his/her code. One is *profiling*, allowing a detailed analysis of which statements are most expensive to execute in a given application (see Section 20.1.5). Another is *evaluation strategies* (see Section 4.4.7), giving the user the possibility of indicating which arguments and in which order to evaluate before simplifying a given operator with the equations. This can range from "no

[6] In its fullest generality, it is well known that associative-commutative rewriting is an NP-complete problem. In programming practice, however, the patterns used as lefthand sides allow much more efficient matching, so that the theoretical limits only apply to "pathological" patterns not encountered in typical programming practice.

arguments" (a lazy strategy) to "all arguments" (an eager bottom-up strategy) to something in the middle (like evaluating the condition before simplifying an if-then-else expression). Evaluation strategies control the positions in which *equations* can be applied. But what about rules? The analogous feature for rules is that of *frozen argument positions*; that is, declaring certain argument positions in an operator with the `frozen` attribute (see Section 4.4.9) blocks rule rewriting anywhere in the subterms at those positions. A fourth useful feature is *memoization* (see Section 4.4.8). By giving an operator the `memo` attribute, Maude stores previous results of function calls to that symbol. This allows trading off space for time, and can lead in some cases to drastic performance improvements.

One nagging question may be reflection. Is reflection really practical from a performance perspective? The answer is yes. In Maude, reflective computations are performed by *descent functions* that move metalevel computations to the object level whenever possible (see Section 17.6). This, together with the use of caching techniques, makes metalevel computations quite efficient. A typical metalevel computation may perform millions of rewrites very efficiently at the object level, paying a cost (linear in the size of the term) in changes of representation from the metalevel to the object level and back only at the beginning and at the end of the computation.

1.2 The logical foundations of Maude

The foundations of a house do not have to be inspected every day: one is grateful that they are there and are sound. This section describes the logical foundations of Maude in an informal, impressionistic style, not assuming much beyond a cocktail party acquaintance with logic and mathematics. The contents of this section may be read in two ways, and at two different moments:

- before reading the rest of the manual, to obtain a bird's-eye view of the mathematical ideas underlying Maude's design and semantics; or
- after reading the rest of the manual, to gain a more unified understanding of the language's design philosophy and its foundations.

Readers with a more pragmatic interest may safely skip this section, but they may miss some of the fun.

Maude is a declarative language in the strict sense of the word. That is, a Maude program is a *logical theory*, and a Maude computation is *logical deduction* using the axioms specified in the theory/program. But which logic? There are two, one contained in the other. The seamless integration of the functional world within the broader context of concurrent, nondeterministic computation is achieved at the language level by the inclusion of functional modules as a special case of system modules. At the mathematical level this inclusion is precisely the sublogic inclusion in which *membership equational logic* [106, 17] is embedded in *rewriting logic* [102, 19].

A functional module specifies a *theory* in membership equational logic. Mathematically, we can view such a theory as a pair $(\Sigma, E \cup A)$. Σ, called the *signature*, specifies the type structure: sorts, subsorts, kinds, and overloaded operators. E is the collection of (possibly conditional) equations and memberships declared in the functional module, and A is the collection of equational attributes (`assoc`, `comm`, and so on) declared for the different operators. Computation is of course the efficient form of equational deduction in which equations are used from left to right as simplification rules.

Similarly, a system module specifies a *rewrite theory*, that is, a theory in rewriting logic. Mathematically, such a rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$, where $(\Sigma, E \cup A)$ is

the module's equational theory part, ϕ is the function specifying the frozen arguments of each operator in Σ, and R is a collection of (possibly conditional) rewrite rules. Computation is rewriting logic deduction, in which equational simplification with the axioms $E \cup A$ is intermixed with rewriting computation with the rules R.

We can of course view an equational theory $(\Sigma, E \cup A)$ as a degenerate rewrite theory of the form $(\Sigma, E \cup A, \phi_\emptyset, \emptyset)$, where $\phi_\emptyset(f) = \emptyset$, that is, no argument of f is frozen, for each operator f in the signature Σ. This defines a sublogic inclusion from membership equational logic (MEqLogic) into rewriting logic (RWLogic) which we can denote

$$\text{MEqLogic} \hookrightarrow \text{RWLogic}.$$

In Maude this corresponds to the inclusion of functional modules into the broader class of system modules. However, Maude's inclusion is more general: the user can give the desired freezing information for each operator in the signature of a functional module, not just the $\phi_\emptyset$ above.

Another important fact is that each Maude module specifies not just a theory, but also an *intended mathematical model*. This is the model the user has intuitively in mind when writing the module. For functional modules such models consist of certain sets of data and certain functions defined on such data, and are called *algebras*. For example, the intended model for a `NAT` module is the natural numbers with the standard arithmetic operations. Similarly, a module `LIST-QID` may specify a data type of lists of quoted identifiers, and may import `NAT` and `BOOL` as submodules to specify functions such as `length` and `_in_`. Mathematically, the intended model of a functional module specifying an equational theory $(\Sigma, E \cup A)$, with Σ the signature defining the sorts, subsorts, and operators, E the equations and memberships, and A the equational attributes like `assoc`, `comm`, and so on, is called the *initial algebra* of such a theory and is denoted $T_{\Sigma/E \cup A}$.

In a similar way, a system module specifying a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$ has an *initial model*, denoted $\mathcal{T}_\mathcal{R}$, which in essence is an algebraic *(labeled) transition system*.[7] The states and data of this system are elements of the underlying initial algebra $T_{\Sigma/E \cup A}$. The state transitions are the (possibly complex) *concurrent rewrites* possible in the system by application of the rules R. For our bank accounts example, these transitions correspond to all the possible concurrent computations that can transform a given "soup" of account objects and messages into another soup. Again, this is the model the programmer of such a system has in mind.

How do the mathematical models associated with Maude modules and the computations performed by them fit together? Very well, thanks. This is the so-called agreement between the *mathematical semantics* (the models) and the *operational semantics* (the computations). In this introduction we must necessarily be brief; see Sections 4.6 and 4.7 and [17] for the whole story in the case of functional modules, and Section 5.3 and [138] for the case of system modules. Here is the key idea: under certain executability conditions required of Maude modules, both semantics coincide. For functional modules we have already mentioned that the equations should have good properties as simplification rules, so that they evaluate each expression to a single final result. Technically, these are called the *Church-Rosser* and *termination* assumptions. Under these assumptions, the final values, called the *canonical forms*, of all expressions form an algebra called the *canonical term algebra*. By definition, the results of operations in this algebra are exactly those given by the Maude interpreter: this is as computational a model as one can possibly get. For example, the results in the canonical term algebra of the operations

```
length('a . 'b . 'c . nil)
'b in ('a . 'b . 'c . nil)
```

[7] With additional operations, including a sequential composition operation for labeled transitions.

are, respectively,

```
s s s 0
true
```

Suppose that a functional module specifies an equational theory $(\Sigma, E \cup A)$ and satisfies the Church-Rosser and termination assumptions. Let us then denote by $Can_{\Sigma/E\cup A}$ the associated canonical term algebra. The coincidence of the mathematical and operational semantics is then expressed by the fact that we have an isomorphism

$$T_{\Sigma/E\cup A} \cong Can_{\Sigma/E\cup A}.$$

In other words, except for a change of representation, both algebras are identical.

For system modules, the executability conditions center around the notion of *coherence* between rules and equations (see [138] and Section 5.3). The equational part $E \cup A$ should be Church-Rosser and terminating as before. A reasonable strategy (the one adopted in Maude by the `rewrite` command, see Chapter 5) is to first apply the equations to reach a canonical form, and then do a rewriting step with a rule in R. But is this strategy *complete*? Couldn't we miss rewrites with R that could have been performed if we had not insisted on first simplifying the term to its canonical form with the equations? Coherence guarantees that this kind of incompleteness cannot happen (see Section 5.3).

1.3 Programming, specification, and verification

The observations in the previous section about the agreement between mathematical and operational semantics in Maude programs are of enormous importance for reasoning about them and verifying their correctness. The key point is that there are three different uses of Maude modules:

1. As *programs*, to solve some application. In principle we could have programmed such an application in some other programming language, but we may have chosen Maude because its features make the programming task easier and simpler.

2. As *formal executable specifications*, that provide a rigorous mathematical model of an algorithm, a system, a language, or a formalism. Because of the agreement between operational and mathematical semantics, this mathematical model is at the same time *executable*. Therefore, we can use it as a precise prototype of our system to *simulate* its behavior. The system itself could be implemented in a conventional language, or perhaps in Maude itself (as in (1) above) as a more detailed Maude program, or maybe our specification is already detailed and efficient enough to be directly used as *its own implementation*.

3. As models that can be *formally analyzed and verified* with respect to different *properties* expressing various *formal requirements*. For example, we may want to prove that our Maude module terminates; or that its equations have the Church-Rosser property; or that a given function, equationally defined in the module, satisfies some properties expressed as first-order formulas. Similarly, given a system module we may want to *model check* some properties about it, such as the satisfaction of some invariants or, more generally, of some temporal logic formulas.

Note that the distinction between uses (1) and (2) is, for the most part, in the eyes of the beholder. In fact, there is a seamless integration of specifications and code. The same

Maude module can simultaneously be viewed as an executable formal specification and as a program. Furthermore, certain kinds of formal requirements needed for verification in (3) can be expressed at the Maude level, either in Maude *theories* (see Section 6.3.1), or by including some *nonexecutable* statements in a Maude module giving them the `nonexec` attribute (see Section 4.5.3). This can be very useful in several ways. For example, we may include *lemmas* that we have proved about a module, either in theories or as nonexecutable statements in the module itself. Similarly, we may begin with some nonexecutable specifications in a Maude theory, and then refine them using *views* (see Section 6.3.2) into the desired Maude module satisfying them.

There is, however, no need for all the properties that we wish to formally verify in (3) to be in the logic of Maude, that is, to be statements in membership equational logic or in rewriting logic. More generally, properties can be expressed, for example, as arbitrary first-order logic formulas, or as temporal logic formulas. An interesting issue is then to explain precisely what it means for a Maude module, defined in membership equational logic or in rewriting logic, to *satisfy* a formula in one of those logics. Here is where the Maude initial model semantics explained in Section 1.2 becomes crucial. Such a semantics means that what a Maude module *denotes* is a specific *mathematical model*, namely, the initial one. Satisfaction of any property, expressed as some kind of formula, means satisfaction of that formula in the initial model. This is an important observation, even when the formula in question is expressed in Maude's native logic. Let us explain this idea in more detail.

Consider, for example, that we have defined natural number addition in a Maude functional module with Peano notation, so that zero is represented as the constant `0`, and there is a successor function `s_` so that, for example, 2 is represented as `s s 0`. Natural number addition can then be defined by the equations

```
op _+_ : Nat Nat -> Nat .
vars N M K : Nat .
eq N + 0 = N .
eq N + (s M) = s (N + M) .
```

The *initial model* of these equations is precisely the algebra of the natural numbers with zero, successor, and the usual addition function. For example, using the canonical term algebra representation (see Section 1.2), when we add `s s 0` and `s s 0` in this algebra we obtain the result `s s s s 0`.

Consider now two relevant *properties* of natural number addition, namely, associativity and commutativity. These properties are precisely described by the respective equations

```
eq N + M = M + N [nonexec] .
eq N + (M + K) = (N + M) + K [nonexec] .
```

where we have used the `nonexec` attribute to emphasize that these equations are not part of our natural number addition module, and are not meant to be executed (in fact, if executed the first equation would loop). They may, for example, be stated in a separate Maude theory as properties we wish to verify.

The first thing to observe is that the above associativity and commutativity equations *are not provable* by equational deduction, that is, they do not follow by replacing equals by equals from the two equations defining the addition function. They are in fact *inductive* properties of the addition function. Therefore, in order to prove them, using for example Maude's inductive theorem prover (ITP), we need to use a stronger proof method, namely, Peano induction. But for any equational specification, being an inductive property and being a property satisfied by its initial model are one and the same thing [111]. Therefore, what we mean when we say that our natural number addition module *satisfies* the associativity and commutativity equations is *precisely* that its initial model does.

Of course, associativity and commutativity are properties expressible in Maude's native logic (in fact, just in its equational sublogic). But the case of arbitrary first-order formulas is entirely similar. Consider, for example, the property that any even number is the sum of two odd numbers, which can be expressed as the first-order formula

$$\forall n : Nat\ (even(n) \implies \exists x, y : Nat\ (odd(x) \wedge odd(y) \wedge n = x + y)).$$

Let us assume, for argument's sake, that we had also defined the *odd* and *even* predicates in our Maude natural number module. What does it mean for our module to satisfy the above formula? Just as before, it exactly means that the initial model denoted by our Maude specification satisfies the formula. The point is that membership equational logic is a sublogic of many-kinded first-order logic with equality (MKFOL$^{=}$) that we can represent with a sublogic inclusion

$$\text{MEqLogic} \hookrightarrow \text{MKFOL}^{=}.$$

Therefore, our initial model is also a first-order logic model, and it is perfectly clear what it means for it to satisfy a first-order formula.

In a similar way, if we have a Maude system module and choose an initial state for it, we may be interested in verifying that it satisfies a given *temporal logic* formula. Defining satisfaction in this case is not as direct as for first-order formulas, because we do not have a sublogic inclusion from rewriting logic into temporal logic. However, the meaning of satisfaction in this case is also fairly straightforward. The point is that to such a system module, that is, to a rewrite theory in which we have defined some atomic state predicates equationally, we can naturally associate a *Kripke structure* (see Section 12.2). Since Kripke structures are the standard models of temporal logic, satisfaction of the given temporal logic formula exactly means that the Kripke structure associated to the module satisfies the formula. In fact, such a Kripke structure and the initial model of the rewrite theory are intimately related, so that the initial model can be used to define the corresponding Kripke structure. As explained in Chapters 11 and 12, if our system module is such that the set of states reachable from the initial state is finite, we can use Maude's `search` command and Maude's model checker for linear temporal logic (LTL) as *decision procedures* to verify, respectively, the satisfaction of invariants and of LTL properties.

Besides being able to use Maude's inductive theorem prover (ITP) to verify inductive properties of functional modules, and the above-mentioned built-in support for verifying invariants and LTL formulas through the `search` command and Maude's LTL model checker, we can use the following Maude tools to formally verify other properties:

- the Maude Termination Tool (MTT) [47, 46] can be used to prove termination of functional modules (see Section 11.4);
- the Maude Church-Rosser Checker (CRC) [54, 56, 30, 48] can be used to check the Church-Rosser property of unconditional functional modules (see Section 11.4);
- the Maude Coherence Checker (ChC) [55, 56] can be used to check the coherence (or ground coherence) of unconditional system modules (see Section 11.4); and
- the Maude Sufficient Completeness Checker (SCC) [83] can be used to check that defined functions have been fully defined in terms of constructors (see Sections 4.4.3 and 11.4).

These tools are integrated in what is called the Maude Formal Environment (MFE), available at `https://github.com/maude-team/MFE`. Furthermore, if we are dealing with rewriting logic specifications of real-time and hybrid systems, we can use the Real-Time Maude tool to both simulate such specifications and to perform search and model-checking analysis of their LTL properties [121, 122].

In summary, therefore, Maude supports three seamlessly integrated tasks: programming, executable formal specification, and formal analysis and verification. For analysis and verification purposes, the Maude interpreter itself is the first and most obvious tool. It is in fact a high-performance *logical engine* that can be used to prove certain kinds of logical facts about our theories. For example, we can use the Maude interpreter as a decision procedure for equational deduction if the desired theory has good properties. Similarly, as already mentioned, we can use it also to verify invariants and LTL properties of finite-state system modules. More generally, we can use other tools in Maude's formal environment, such as the ITP, MTT, CRC, ChC, and SCC tools (or Real-Time Maude for real-time systems) to formally verify a variety of other properties.

1.4 A high-performance logical framework

Our previous discussion of the programming, executable specification, and formal verification uses of Maude makes clear that we can distinguish two different levels of formal specification: a *system specification* level, and a *property specification* one. In a system specification we are after an unambiguous specification of a given system and how it actually *works*. Ideally this specification should be both formal and executable, and should therefore provide an *executable mathematical model* of the system we are interested in. This is exactly what Maude modules provide.

By contrast, when specifying *properties* of a system we are not necessarily after an executable model of our system. Instead, we *assume it*, as either already given or to be developed later, and specify such properties in a typically nonexecutable manner: for example in first-order logic, higher-order logic, or some temporal logic. That is, the properties we specify have an *intended model*, namely the system design captured by a system specification, and we are interested in *verifying* by different methods that the intended model *satisfies* the properties stated in our property specification. In the context of Maude, such property specifications can be given in a variety of ways:

- as nonexecutable equations, memberships, and rules in Maude's native logics;
- as first-order logic formulas; or
- as invariants or, more generally, linear temporal logic formulas.

We can then use Maude itself and its formal tool environment to try to verify that a given system specified as a Maude module satisfies the desired properties.

Since Maude system specifications should be both formal and executable, Maude native logics, namely, membership equational logic and its rewriting logic extension, should be *computational* logics, that is, logics in which computation and deduction coincides, and simple enough to allow a *high-performance* implementation as a declarative programming language. This is what the Maude implementation provides. Of course, as mentioned in Section 1.2 and further explained in Sections 4.6 and 5.3, Maude modules should be theories that satisfy some reasonable *executability requirements*, making possible not only their efficient execution, but also the already-mentioned coincidence between mathematical and operational semantics.

However, not all computational logics are equally *expressive*. For example, equational logics (in either first-order or higher-order versions) are very well suited to specify *deterministic systems* under the Church-Rosser assumption, but poorly equipped to specify concurrent and highly nondeterministic systems. The whole point of extending membership equational logic to rewriting logic is to seamlessly integrate the specification of deterministic systems, through

equational specifications in functional modules, and of concurrent and nondeterministic systems, through rewriting logic specifications in system modules, within the same language. Experience has shown that this makes rewriting logic a very expressive *semantic framework* for system specification. Here we only mention some relevant areas:

- *Models of computation.* Many models of computation, including a very wide range of concurrency models, can be naturally specified as different theories within rewriting logic, and can be executed and analyzed in Maude.

- *Programming languages.* Rewriting logic has very good properties—combining in a sense the best features of denotational semantics' equational definitions with the strengths of structural operational semantics—to give formal semantics to a programming language. Furthermore, in Maude such semantics definitions become the basis of interpreters, model checkers, and other program analysis tools for the language in question.

- *Distributed algorithms and systems.* Because of its good features for concurrent, object-based specification, many distributed algorithms and systems, including, for example, network protocols and cryptographic protocols, can be easily specified and analyzed in Maude. Furthermore, making use of Maude's external object facility to program interactions with internet sockets, one can not just specify but also program various distributed applications in a declarative way (see Section 9).

- *Biological systems.* Cell dynamics is intrinsically concurrent, since many different biochemical reactions happen concurrently in a cell. By modeling such biochemical reactions with rewrite rules, one can develop useful symbolic mathematical models of cell biology. Such models can then be used to study and predict biological phenomena.

Furthermore, other application areas can be naturally supported in appropriate *extensions* of rewriting logic and Maude. For example, real-time and hybrid systems can be specified as *real-time rewrite theories*. Such specification can be executed and analyzed in the Real-Time Maude tool [121, 122]. Similarly, probabilistic systems can be specified as *probabilistic rewrite theories*, and can be simulated in PMaude and analyzed in the VeStA tool [90, 2].

The fact that in a computational logic computation and deduction coincide, so that they are like two sides of the same coin, can be used in two ways: we can use the logic as a semantic framework to specify different computational entities as just explained; or we can use it as a *logical framework* to represent many other logics in it. That is, if our computational logic has good representational features, it can be used as a *universal logic* which can faithfully express the inference systems of many other logics.

Since the logic is computational and presumably has an efficient implementation, this is not just a purely theoretical exercise: we can use such an implementation to *mechanize* deduction in any logic that we can faithfully represent inside our logical framework. Experience has shown that rewriting logic has very good properties as a logical framework in precisely this sense. An important practical consequence is that it becomes quite easy to use Maude to develop a variety of *formal tools* for different logics. The point is that any such tool has an associated inference system, so it is just a matter of representing such an inference system as a rewrite theory and guiding the application of the inference rules with suitable strategies (see Section 17.7). In addition, since such formal tools often manipulate and transform not only formulas but also *theories*, Maude's reflective capabilities, which allow manipulating theories as data, become enormously useful [30, 31].

Reflection and the existence of initial models (and therefore of induction principles for such models) have one further important consequence, namely, that rewriting logic has also good

properties as a *metalogical framework*. A metalogical framework is a logical framework in which we can not only represent and simulate many other logics: we can also *reason* within the framework about the metalogical properties of the logics thus represented. As explained in [9], this is exactly what can be done in rewriting logic using Maude and Maude's inductive theorem prover (ITP).

1.5 Core Maude vs. Full Maude

We call *Core Maude* the Maude 2 interpreter implemented in C++ and providing all of Maude's basic functionality. Part I explains in detail all the aspects of Core Maude, including its syntax and parsing, functional and system modules, module hierarchies, module parameterization with theories and module instantiation with views, its suite of predefined modules, the model-checking capabilities, object-based programming, reflection, and metalanguage uses.

Full Maude is an extension of Maude, written in Maude itself, that endows the language with an even more powerful and extensible *module algebra* than that available in Core Maude. As in Core Maude, modules can be parameterized and instantiated with views, but in addition views can also be parameterized. Full Maude also provides generic modules for n-tuples. Object-oriented modules (which can also be parameterized) support notation for objects, messages, classes, and inheritance.

Full Maude itself can be used as a basis for further extensions, by adding new functionality. It is possible both to change the syntax or the behavior of existing features, and to add new features. In this way Full Maude becomes a common infrastructure on top of which one can build tools, such as, e.g., the Church-Rosser and coherence checkers, as well as environments for other languages, such as, e.g., the Real-Time Maude tool for specifying and analyzing real-time systems [121, 122], and the Maude MSOS tool for modular structural operational semantics [22].

1.6 Manual structure

The present manual documents Maude 3, and explains Maude's basic concepts in a leisurely and mostly informal style. The material is basically presented following a "grammatical" order; for example, all features related with operators are discussed together. Concepts are introduced by concrete examples, that may be fragments of modules. The complete module examples are available in the official webpage `http://maude.cs.illinois.edu`. We follow the convention of naming each file in that site as the module it contains, or if a file contains several modules, as the main one. In most cases, the rewriting or search commands and the corresponding outputs are also included in the files.

The manual is divided in three parts: Part I is devoted to Core Maude, Part II is devoted to Full Maude, and Part III is a reference manual. Here is a brief summary of what can be found in the remaining chapters:

Part I. Core Maude.

Chapter 2 explains how to get Maude, how to install the system on the different platforms supported, and how to run it. It also includes pointers on how to get additional information and support.

Chapter 3 describes the basic syntactic constructs of the language, including what is an identifier, a sort, and an operator. The different kinds of declarations that can

be included in the different types of modules are explained here, in addition to fundamental concepts such as kinds or terms, and a discussion on parsing.

Chapter 4 introduces *functional modules*, and the different statements that can be found in this kind of modules, namely equations and membership axioms. Operator and statement attributes are also introduced. The final part of this chapter is devoted to the use of functional modules for equational simplification, for which matching modulo axioms is a fundamental feature.

Chapter 5 introduces *system modules*, and is mainly devoted to rules, term rewriting, and the `search` command.

Chapter 6 explains the support for modularity provided by Core Maude. It describes first the different modes of module importation, namely *protecting*, *extending*, and *including*. Then it introduces the module summation and renaming operations. Finally, this chapter explains the powerful form of parameterized programming available in Core Maude, based on theories and views.

Chapter 7 provides detailed descriptions of the different predefined data types available, including Booleans, natural numbers, integers, rationals, floating-point numbers, strings, and quoted identifiers. It also describes the generic containers provided by Maude, namely lists, sets, maps, and arrays. The chapter finishes with a description of a built-in linear Diophantine equation solver.

Chapter 8 explains the basic support for object-based programming, with special emphasis on the standard notation for object systems.

Chapter 9 explains Maude's support for rewriting with external objects and the implementation of sockets, standard streams, and files.

Chapter 10 describes Maude's object-level strategy language and the corresponding strategy modules. It contains many examples of using strategies to control rewriting.

Chapter 11 explains how to use the `search` command to model check invariant properties of concurrent systems specified as system modules in Maude.

Chapter 12 introduces linear temporal logic (LTL) and describes the facilities for LTL model checking provided by the Maude system. This procedure can be used to prove properties when the set of states reachable from an initial state in a system module is finite. When this is not the case, it may be possible to use an equational abstraction technique for reducing the size of the state space.

Chapter 13 describes Maude's support of order-sorted unification modulo axioms such as either commutativity or associativity and commutativity. The importance of this feature is made explicit in an overview of several interesting applications of unification, including narrowing and symbolic reachability analysis. This chapter also includes a discussion on endogenous vs. exogenous order-sorted unification algorithms.

Chapter 14 introduces the concept of *variants*, and then explains its use in the `get variants` command and its application in variant-based equational order-sorted unification.

Chapter 15 describes the implementation of narrowing based on the unification and variant facilities introduced in the previous Chapters 13 and 14.

Chapter 16 introduces *satisfiability modulo theories* (SMT) and describes the connection of Maude to SMT solvers like CVC4 and Yices2.

Chapter 17 presents the reflective capabilities of the Maude system. The concept of *reflection* is introduced, and the effective way of supporting metalevel computation is discussed. The predefined module `META-LEVEL` and its submodules are presented, with special emphasis on the descent functions provided. The chapter ends with an introduction to the notion of internal strategies.

Chapter 18 explains the way of using the facilities provided by the modules `META-LEVEL`, `STD-STREAM`, and `LEXICAL` for the construction of user interfaces and metalanguage applications.

Chapter 19 introduces meta-interpreters. Each meta-interpreter is an external object that is an independent Maude interpreter, complete with module and view databases, and able to send and receive messages. Together with standard streams, meta-interpreters can be used to develop execution environments.

Chapter 20 discusses debugging and troubleshooting, considering the different debugging facilities provided: tracing, term coloring, the debugger, and the profiler. A number of traps and known problems are also commented.

Part II. Full Maude.

Chapter 21 explains the nature of Full Maude, and how to use it. This chapter includes information on how to load Core Maude modules into Full Maude, on the additional module operations (supported by tuple generation and parameterized views), and on the facilities available in Full Maude for moving up and down between reflection levels.

Chapter 22 introduces object-oriented modules, which provide a syntax more convenient than that of system modules for object-oriented applications, with direct support for the declaration of classes, inheritance, and useful default conventions in the definition of rules. Such object-oriented modules can also be parameterized. This chapter includes several extended examples that illustrate the power of combining the additional features available in Full Maude.

Part III. Reference.

Chapter 23 gives a complete list of the commands available in Maude.

Chapter 24 includes the grammar of Core Maude.

1.7 The Maude book

Most of the material in this manual also appears in the book *All About Maude: A High-Performance Logical Framework*, published by Springer as volume 4350 in the series Lecture Notes in Computer Science [29].

The book contains many additional examples and explanations, as well as information on applications and tools.

We gratefully acknowledge the permission given by Springer to distribute this manual on the web.

Acknowledgements

Languages are living organisms. The lifeblood provided by experienced users is key to their growth and their improvement. We have benefited much from colleagues who have used different alpha versions of Maude; we cannot mention them all, but Luis Aguirre, Christiano Braga, Feng Chen, Grit Denker, Santiago Escobar, Azadeh Farzan, Joe Hendrix, Merrill Knapp, Nirman Kumar, Óscar Martín, Miguel Palomino, Peter Ölveczky, José Quesada, Adrián Riesco, Dilia Rodriguez, Grigore Roşu, Ralf Sasse, Koushik Sen, Ambarish Sridharanarayanan, Mark-Oliver Stehr, Prasanna Thati, and Alberto Verdejo deserve special thanks for their creative uses of Maude and their suggestions for improving the language. Thanks to Christiano Braga, Peter Mosses, Peter Ölveczky, Miguel Palomino, Sylvan Pinsky, Isabel Pita, Adrián Riesco, Dilia Rodriguez, Manuel Roldán, Mark-Oliver Stehr, Antonio Vallecillo, and Alberto Verdejo for their comments on previous versions of this document.

As already mentioned, Maude's historical precursor is the OBJ3 language [79]. The OBJ3 experience has greatly influenced the Maude design and philosophy, and we are grateful to all our former OBJ colleagues for this. Joseph Goguen should be mentioned in particular, because of his enormous influence in all aspects of OBJ; and Tim Winkler for having implemented a state-of-the-art OBJ3 system with such great skill.

Two other rewriting logic languages, ELAN [12] and CafeOBJ [75], have provided a rich stimulus to the design of Maude. Although our language design solutions have often been different, we have all been wrestling with a similar problem: how to best obtain efficient language implementations of rewriting-based languages. We have benefited much from the ELAN and CafeOBJ experience, and from many discussions with their main designers and implementers: Claude and Hélène Kirchner, Marian Vittek, Pierre-Etienne Moreau, Kokichi Futatsugi, Râzvan Diaconescu, Ataru Nakagawa, Toshimi Sawada, and Makoto Ishisone.

Bringing a new language design to maturity requires a long-term research effort and substantial resources. We are not there yet, but much has been advanced since the early design phases. Perhaps the longest, most sustained support has come from the US Office for Naval Research (ONR) through a series of contracts. We are most grateful to Dr. Ralph Wachter at ONR for his continued encouragement at every step of the way. The US Defense Advance Research Projects Agency (DARPA), the US National Science Foundation (NSF), and the Spanish Ministry for Education and Science (MEC) have also contributed important resources to the development of Maude, its foundations, and its applications.

Part I

Core Maude

Chapter 2

Using Maude

2.1 Getting Maude

The Maude system is available, free of charge, under the terms of the GNU General Public License as published by the Free Software Foundation, at the Maude home page (a snapshot is shown in Figure 2.1)

```
http://maude.cs.illinois.edu
```

There you can also find documentation about Maude, including a Maude primer, some papers on Maude and rewriting logic, and several Maude applications, including a set of proving tools for Maude specifications and Maude case studies.

Maude binaries are provided for selected architectures and operating systems, including Linux and macOS. Detailed information on this can be found in the Maude web site, where installation instructions are also available.

2.2 Running Maude

A Maude session can be started by calling the `maude.linux64` binary included in the release package in a Linux shell window (and similarly for other platforms). For example, we can move into the directory where the package was extracted and then invoke Maude, obtaining a banner similar to the following, where we can see the version of the system being executed, the date it was built, copyright information, and the current date.

```
~/maude-linux$ ./maude.linux64
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 3.1 built: Oct 12 2020 20:12:31
            Copyright 1997-2020 SRI International
                  Tue Oct 13 12:00:00 2020
Maude>
```

The Maude system is now ready to accept Maude modules and commands (see Chapter 23 for a complete list of Maude commands). During a Maude session, the user interacts with the system by entering a request at the Maude prompt. For example, one can quit:

```
Maude> quit
```

Figure 2.1: Maude home page at `maude.cs.illinois.edu`

`q` may be used as an abbreviation of the `quit` command. But please, do not leave us so soon! One can also enter modules and use other commands. For example, we can enter the following module `SIMPLE-NAT`, which specifies the natural numbers in Peano notation with a plus operation `_+_` on them.[1]

```
Maude> fmod SIMPLE-NAT is
         sort Nat .
         op zero : -> Nat .
         op s_ : Nat -> Nat .
         op _+_ : Nat Nat -> Nat .
         vars N M : Nat .
         eq zero + N = N .
         eq s N + M = s (N + M) .
       endfm
```

Fortunately, we do not need to write our modules in the prompt. We can input one or several modules by saving them in a file and then entering the file with the `in`, `load` or `sload` commands (see Section 23.16 for details on the difference between these commands). Assuming that the file `my-nat.maude` contains the module `SIMPLE-NAT` above, we can do the following to enter it:

```
Maude> load my-nat.maude
```

After entering the module `SIMPLE-NAT` into Maude, we can, for example, reduce the term `s s zero + s s s zero` (which is the equivalent in Peano notation of the more usual $2 + 3$) as follows:

```
Maude> reduce in SIMPLE-NAT : s s zero + s s s zero .
reduce in SIMPLE-NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Nat: s s s s s zero
```

It is not necessary to give the name of the module in which to reduce a term explicitly. All commands that require a module refer to the current module by default, unless a module is explicitly given. The current module is usually the last module entered or used, although we can use the `select` command to select a module to be the current one (see Section 23.15).

```
Maude> reduce s s zero + s s s zero .
reduce in SIMPLE-NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Nat: s s s s s zero
```

Any action happening in the Maude system can be interrupted by typing control-C. In particular, by hitting control-C during a reduction in progress, such reduction is interrupted and the system gets into debugging mode (see Section 20.1.3).

Although it is not the case in the simple examples above, sometimes we get a very big term as output from Maude. In some cases, in order to make it easier to read and understand, we edit the presentation of the outputs given by Maude.

When you execute `maude.linux64`, the file `prelude.maude`, which includes several predefined modules (see Chapter 7), is automatically loaded. To find `prelude.maude`, the Maude interpreter checks for it in several directories, in the following order:

1. the directories specified in the `MAUDE_LIB` environment variable,

2. the directory containing the executable, and

[1] We do not display the '`>`' symbol that Maude adds at the beginning of each line.

3. the current directory.

It is a good idea to include the path to `prelude.maude` in the `MAUDE_LIB` environment variable to be sure that it will always be found, because the executable finding code may not find the directory containing the executable.

Among the predefined modules included in `prelude.maude` we find a module `STRING` that declares sorts and operations for manipulating strings. In particular, `STRING` introduces the operation `_+_` to concatenate two strings. Then, to concatenate the strings "hello", " ", and "world", you can type at the Maude prompt the following `red` (which is the abbreviated form of `reduce`) request:

```
Maude> red in STRING : "hello" + " " + "world" .
reduce in STRING : "hello" + " " + "world" .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result String: "hello world"
```

Actually, although `STRING` is not the current module right after starting the system, it is imported by the current one, `CONVERSION`. Thus, we can type the following, just after starting Maude:

```
Maude> red "hello" + " " + "world" .
reduce in CONVERSION : "hello" + " " + "world" .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result String: "hello world"
```

Notice that Maude makes explicit the module in which the term is reduced, even when no module name is given by the user.

As said above, to load for example a user-defined module `HELLO-WORLD` for a Maude session, you can either type at the Maude prompt the whole module or simply type the following `in`-troduce request:

```
Maude> in hello-world
```

where `hello-world` is a text file in the current directory containing the module `HELLO-WORLD`.

For files specified by a bare file name, Maude also checks for files with `.maude`, `.fm`, and `.obj` extensions. Maude can also be told using the `MAUDE_LIB` environment variable about other directories to use to search for files. Thus to find a file specified in the `in` command, Maude searches, in order:

1. the current directory,

2. the directories in the `MAUDE_LIB` environment variable, and

3. the directory containing the executable.

If the desired file is in none of these places you must type its full path name.

As for user-defined modules, user requests such as the above can either be typed at the Maude prompt or simply `in`-troduced with a text file containing them. In fact, many users run Maude inside an Emacs-like editor, since this provides both text editing facilities for creating Maude modules and saving them in files, and also UNIX shell interaction to start a Maude session and to `in`-troduce during the session modules and commands created and saved in files, as shown in Figure 2.2.

Note that text files entered in Maude can contain not only modules, but also any command. Actually, a file can contain as many modules and commands as one wishes. When entering it with an `in` or `load` command, Maude will read them one after another as if they were written

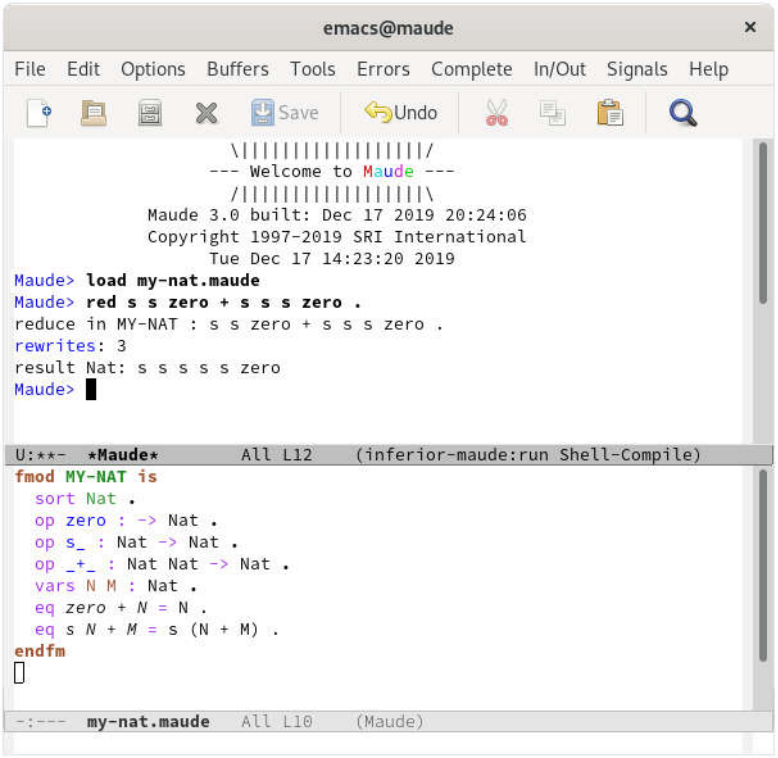


Figure 2.2: Running Maude inside Emacs

at the prompt of the system. Another important issue worth pointing out is that we can write single line and multiline comments anywhere inside a module or a file. Single line comments are started by either `***` or `---`, and ended by the end of line. Multiline comments are started by `***(` and ended by `)`. Parentheses must balance within multiline comments.

2.3 Getting support and more information

We maintain the following mailing lists related to Maude:

- `maude-users@maude.cs.illinois.edu`. A moderated list for the discussion of topics of general interest to all Maude users. This list is typically low-traffic, and contains items such as calls for papers, announcements of new Maude related papers, and notifications of new releases of Maude. It is important that you subscribe to this list if using Maude, as this is the mechanism by which we will make important announcements about the system. To subscribe, or to view the archived messages, please go to

  `https://lists.cs.illinois.edu/lists/info/maude-users`

- `maude-help@maude.cs.illinois.edu`. This is an alias for submitting questions about any aspect of the use of Maude. Messages are distributed to a group of experienced users who have offered to provide help. This list is not open for subscription, but you can send

messages to this list at any time. Questions posted here will be automatically archived at

`https://lists.cs.illinois.edu/lists/info/maude-help`

- `maude-bugs@maude.cs.illinois.edu`. A list for reporting any problems you experience with Maude (see below), and also any suggestions for enhancements and improvements.

2.4 Reporting bugs in Maude

As already mentioned, bug reports should be sent to

`maude-bugs@maude.cs.illinois.edu`

When submitting a bug report, please include the following information:

1. *Example to reproduce the bug.* Ideally this should be a single file that reproduces the bug by loading it. If your example is large and spread out in multiple files, have a file `top.maude` that loads files and executes commands as necessary to reproduce the bug. Send all the files as a tar archive, optionally compressed with gzip.

   If Maude's output is not obviously wrong (for example, an "internal error" message), include an explanation of why the output is wrong.

   If you choose to simplify the example, note that a short runtime to expose the bug is desirable. A small example text is mostly unimportant unless it is necessary to understand such example text in order to understand why Maude's output is incorrect.

2. *Version of Maude used.* Make sure you provide information of the concrete release of Maude (and Full Maude if it is the case). If you are not using one of the ready-made binaries released by the Maude team, also give the versions of the compiler and tools used to build it and the libraries linked against.

3. *Platform.* Include the operating system type and version number, as well as the processor type.

Chapter 3

Syntax and Basic Parsing

This chapter introduces the basic syntactic ingredients of all Maude specifications: identifiers, module names, sort names, and operator declarations. Other syntactic parts of Maude specifications, like equations and rules, will appear in the following chapters.

Some syntax is presented in an informal way by means of general schemes; a formal BNF grammar of the language can be found in Chapter 24.

The chapter finishes explaining some features that can be used to reduce parsing ambiguities in the user-definable syntax, including mixfix operator declarations, supported by Maude.

3.1 Identifiers

In Core Maude, identifiers are the basic syntactic elements, used to name modules and sorts, and to form operator names. For example, `NAT`, `Nat`, and `hello-world` are identifiers. In general, an identifier in Maude is any finite sequence of ASCII characters such that:

- It does not contain any white space. For example, the sequence '`abc def`' is not one identifier, but two.

- The characters '`{`', '`}`', '`(`', '`)`', '`[`', '`]`' and '`,`' are *special*, in that they break a sequence of characters into several identifiers. For example, the sequence `ab{c,d}ef` counts as *seven* identifiers, namely, `ab`, `{`, `c`, `,`, `d`, `}`, and `ef`.

- The backquote character '`` ` ``' is used as an *escape character* to indicate that a blank space or the special characters do not break the sequence. Consequently, backquotes can *only* appear immediately *before* any of the special characters, or *between* two non-empty strings of characters—with neither the ending of the first string nor the beginning of the second string being another backquote—for exactly these purposes. For example, `1`ab`{c`,d`}ef` is a single identifier. Maude's pretty printer will display such an identifier in the form `1 ab{c,d}ef`.

Nonprinting characters in strings use C backslash conventions [87, Section A2.5.2].

3.2 Modules

In Maude the basic units of specification and programming are called *modules*.[1] A module consists of syntax declarations, providing appropriate language to describe the system at hand, and of statements, asserting the properties of such a system. The syntax declaration part is called a *signature* and consists of declarations for:

- *sorts*, giving names for the types of data,
- *subsorts*, organizing the data types in a hierarchy,
- *kinds*, that are implicit and intuitively correspond to "error supertypes" that, in addition to normal data, can contain "error expressions," and
- *operators*, providing names for the operations that will act upon the data and allowing us to build expressions (or terms) referring to such data.

We use symbols Σ, Σ', etc. to denote signatures.

In Core Maude there are two kinds of modules: *functional modules* and *system modules*. Signatures are common for both of them. The difference between functional and system modules resides in the statements they can have:

- functional modules admit *equations*, identifying data, and *memberships*, stating typing information for some data, while
- system modules also admit *rules*, describing transitions between states, in addition to equations and memberships.

We use E, E', etc. to denote sets of equations and memberships, and R, R', etc. to denote sets of rules.

From a programming point of view, a *functional module* is an equational-style functional program with user-definable syntax in which a number of sorts, their elements, and functions on those sorts are defined. From a specification viewpoint, a functional module is an *equational theory* (Σ, E) with initial algebra semantics. Functional modules are described in detail in Chapter 4, here we just discuss some of their top-level syntax. Each functional module has a *name*, which is a Maude identifier. Any Maude identifier can be used, but the preferred style for module names is an all capitalized identifier, and in the case of a compound name the different parts are linked with hyphens. For example, a module defining numbers and operations on them can be called `NUMBERS`. The top-level syntax will then be

```
fmod NUMBERS is
  ...
endfm
```

with '...' corresponding to all the declarations of submodule importations, sorts, subsorts, operators, variables, equations, and so on.

From a programming point of view, a *system module* is a declarative-style concurrent program with user-definable syntax. From a specification viewpoint, it is a *rewrite theory* (Σ, E, ϕ, R) (where ϕ specifies the frozen arguments of operators in Σ; see Section 4.4.9) with initial model semantics. Again, each system module has a *name*, which is a Maude identifier. And as for functional modules, the preferred style is an all capitalized name, with consecutive parts linked with hyphens in the case of compound names. For example, a module specifying

[1] As explained in Section 6.3.1, specifications can also be given in *theories*, with a syntax entirely similar to that of modules, but theories, unlike modules, need not be executable.

the behavior of a vending machine may be called `VENDING-MACHINE`. It will then be introduced with the following top-level syntax:

```
mod VENDING-MACHINE is
  ...
endm
```

where again '...' corresponds to all the declarations of submodule importations, sorts, subsorts, operators, variables, equations, rules, and so on. System modules are described in detail in Chapter 5.

In the rest of the chapter we will describe the ingredients of signatures, that is, the syntactic elements common to both functional and system modules, such as sorts, subsorts, kinds, operators, variables, and the terms that can be built on a signature, postponing the discussion about the syntax specific to functional and system modules to Chapters 4 and 5, respectively.

3.3 Sorts and subsorts

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. Sorts can be partially ordered via a *subsort* relation.

A sort is declared using the `sort` keyword followed by an identifier (the sort name), followed by white space and a period, as follows:

`sort` ⟨*Sort*⟩ `.`

and multiple sorts may be declared using the `sorts` keyword, as follows:

`sorts` ⟨*Sort-1*⟩ `...` ⟨*Sort-k*⟩ `.`

The period at the end of the sort declaration, as for the other types of declarations, is crucial. Note that if either the period is missing or no space is left before and after the period, there can be parsing problems or unintended behavior. For example, the following declaration is syntactically correct but causes an unintended interpretation because of a missing '`.`', since this way sorts `A`, `B`, `sort`, and `C` are declared.

```
sorts A B
sort C .
```

Note also that the keywords `sort` and `sorts` are synonyms. One may use `sort` for multiple sort declarations and `sorts` for single ones, although we do not encourage this style.

For example, we can declare sorts `Zero`, `NzNat`, and `Nat` in the `NUMBERS` module, either one at a time

```
sort Zero .
sort NzNat .
sort Nat .
```

or all at once

```
sorts Zero Nat NzNat .
```

The identifiers `<`, `->`, and `~>` cannot be used as sort names. Moreover, identifiers used for sorts cannot contain any of the characters '`:`', '`.`', '`[`', or '`]`'. The reasons for these restrictions will become clear below in this section and in Sections 3.4, 3.5, and 17.2.1. The use of '`{`', '`}`', and '`,`' is only allowed in structured sort names (see below). Although any so restricted identifier is a legal sort name, the preferred style is to capitalize the *first* letter of the name.

Furthermore, in the case of a compound name, such as a sort of nonzero naturals, the names (each with the first letter capitalized) or suitable abbreviations will be *juxtaposed* without spaces or hyphens, like, for example, `NzNat`.

A sort name can also be structured. Structured sort names are used in parameterized modules; for example, we may use `List{X}` for a parameterized list sort with parameter `X` and `List{Nat}` for its instantiation to lists of natural numbers (see Section 6.3.3). A *structured sort name* contains at least one pair of curly brace symbols '`{`' and '`}`', and is constructed according to the following BNF grammar, without any white space between terminals:

⟨*Sort*⟩ `::=` ⟨*sort identifier*⟩
| ⟨*Sort*⟩ `{` ⟨*SortList*⟩ `}`
⟨*SortList*⟩ `::=` ⟨*Sort*⟩
| ⟨*SortList*⟩ `,` ⟨*Sort*⟩

Notice that structured sorts *are* allowed to contain '`{`', '`,`' and '`}`' but only in accordance with the above grammar. Thus all the following are structured sort names:

```
a{X}
a{X, Y}
a{b, c{d}}{e}
a{(}
```

while the following are *not* legal sort names:

`{X}`	(lacks sort identifier prefix)
`a(X, Y)`	('`,`' not inside braces)
`a{b, {d}}{e}`	(`{d}` lacks sort identifier prefix)
`a({)`	('`{`' without closing '`}`')

Structured sort names can be written in an equivalent *single-identifier form* by using backquotes. For example, the sort `a{b, c{d}}{e}` may be written as `a‘{b‘,c‘{d‘}‘}‘{e‘}`. Hybrid notation such as in `a{b‘,c}` is not allowed. When backquotes are omitted, the sort name becomes a sequence of tokens according to Maude's usual tokenization rules and arbitrary white space may be inserted between tokens. For example, `Foo‘{X‘,Y‘}`, `Foo{X,Y}`, and `Foo{X, Y}` are three equivalent forms for the same structured sort name.

Structured sort names must be written in their equivalent single-identifier form inside operator hooks (see Chapter 7) or in metasyntax (see Chapter 17).

Apart from their special syntax and their use as parameterized sorts in parameterized modules (see Section 6.3.3), structured sort names behave just like sort identifiers.

The subsort relation on sorts parallels the subset relation on the sets of elements in the intended model of these sorts. Subsort inclusions are declared using the keyword `subsort`. The declaration

`subsort` ⟨*Sort-1*⟩ `<` ⟨*Sort-2*⟩ `.`

states that the first sort is a subsort of the second. For example, the declarations

```
subsort Zero < Nat .
subsort NzNat < Nat .
```

specify that the sorts `Zero` (containing only the constant 0) and `NzNat` (the nonzero natural numbers) are subsorts of `Nat`, the natural numbers. More than one subsort relationship can be declared using the keyword `subsorts`, as follows:

`subsorts` ⟨*Sort-1*⟩ `...` ⟨*Sort-j*⟩ `<` `...` `<` ⟨*Sort-k*⟩ `...` ⟨*Sort-l*⟩ `.`

Then, the above declarations can be given in a single declaration as follows:

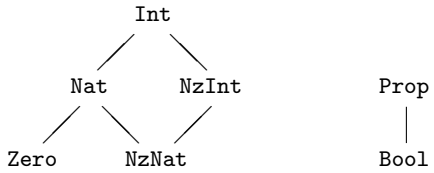
```
subsorts Zero NzNat < Nat .
```

If we extend `NUMBERS` with sorts `Int` and `NzInt` we can express the additional subsort relationships compactly by

```
sorts NzInt Int .
subsorts NzNat < NzInt Nat < Int .
```

A set of subsort declarations must define a *partial order* among the set of sorts. For this to be true, the user is required to avoid *cycles* in the subsort declarations. For example, if a sort `A` is declared as a subsort of `B`, and `B` is declared as a subsort of `A`, we would have a cycle.

Note that the partial order of subsort inclusions partitions the set of sorts into *connected components*, that is, into sets of sorts that are directly or indirectly related in the subsort ordering. For example, all the above sorts `Zero`, `Nat`, `NzNat`, `NzInt`, and `Int` belong to the same connected component in the subsort ordering, whereas a sort `Bool` would clearly belong to a different connected component and could have other sorts, for example a supersort `Prop` of propositions, related to it in the same component. Intuitively, connected components gather together related sorts of data such as numerical data, truth-value data, and so on. Graphically, we can visualize the partial order of subsort inclusions as an acyclic graph (the corresponding *Hasse diagram*), and then the connected components are exactly those of the underlying graph, as in the following example:



3.4 Operator declarations

In a Maude module, an operator is declared with the keyword `op` followed by its *name*, followed by a colon, followed by the list of sorts for its arguments (called the operator's *arity* or *domain sorts*), followed by `->`, followed by the sort of its result (called the operator's *coarity* or *range sort*), optionally followed by an attribute declaration (the discussion of operator attributes is postponed to Section 4.4), followed by white space and a period. Thus the general scheme has the form

`op` ⟨*OpName*⟩ `:` ⟨*Sort-1*⟩ `...` ⟨*Sort-k*⟩ `->` ⟨*Sort*⟩ `[`⟨*OperatorAttributes*⟩`]` `.`

Here are some operator declarations for our `NUMBERS` module.

```
op zero : -> Zero .
op s_ : Nat -> NzNat .
op sd : Nat Nat -> Nat .
ops _+_ _*_ : Nat Nat -> Nat .
```

If the argument list is empty, the operator is called a *constant*. Thus `zero` is a constant.

The name of the operator is a string of characters that may consist of several identifiers, due to the presence of blanks or other special characters. Underscores (`_`) play a special role in these strings. If no underscore character occurs in the operator string—as in the case of the operator `sd` above—then the operator is declared in *prefix* form. If underscore characters occur

in the string, then their number must coincide with the number of sorts declared as arguments of the operator (in particular, constant names cannot include any underscore character). The operator is then in *mixfix* form, with the n-th underscore indicating the place where arguments of the n-th sort must be placed in expressions formed with that operator. In the above example the operators `s_`, `_+_`, and `_*_` are in mixfix form.

There may or may not be any other characters before or after any of the underbars. If no other characters appear, we say that the operator has been declared with *empty syntax*. For example, we could declare a sort `NatSeq` of sequences of natural numbers formed with empty syntax as follows:

```
sort NatSeq .
subsort Nat < NatSeq .
op __ : NatSeq NatSeq -> NatSeq [assoc] .
```

where `assoc` is an attribute declaring that sequence concatenation is associative (see Section 4.4.1). With this operator declaration we can write number sequences such as

```
zero (s zero) (s s zero)
```

Operators having the same arity and coarity can be declared simultaneously by using the keyword `ops` and giving the non-empty list of their corresponding names after the `ops` keyword and before the `:`, as is done for the declarations of `_+_` and `_*_` in the example above.

An operator can also be declared using *several identifiers*. This can be due to the presence of special characters, or to blank spaces, or both. Consider for example the operator declaration

```
op [_] and then [_] : Command Command -> Command .
```

that may allow a natural language style in the syntax of a programming language. It uses eight identifiers in the Maude sense, but declares a single binary operator, with the underscores indicating the place of the arguments in the mixfix notation. Internally, Maude also associates to this operator a corresponding *single-identifier form* by using backquotes. We could have equivalently defined the operator using the single-identifier form, namely,

```
op `[_`]and`then`[_`] : Command Command -> Command .
```

Of course, both variants are equivalent and have the same mixfix display, but the version without backquotes is obviously more convenient.[2]

The declaration of an operator requires an extra pair of parentheses if we already use parentheses as part of the syntax of the operator. Suppose we had in a programming language a binary operator `(_ only after _)`. Then, we have to declare it as follows:

```
op ((_ only after _)) : Command Command -> Command .
```

Since an operator may be declared using several identifiers, in an `ops` declaration involving several operators each operator declaration can be enclosed in parentheses if necessary, to indicate where the syntax of each operator begins and ends. We could have declared both operators together, as follows:

```
ops ([_] and then [_]) ((_ only after _)) :
  Command Command -> Command .
```

Thus, one or several Maude identifiers can be used in operator declarations. Regarding style, the preferred one, particularly for single-identifier operators with prefix syntax, is to use lower case names. However, for a composed name such as a *meta parse* operator, the subsequent names will be juxtaposed and will typically begin with a capital letter to enhance readability, e.g., `metaParse`.

[2] In Full Maude, operator names in operator declarations must be given as single identifiers. Multiple-identifier names are also supported, but their equivalent single-identifier form must be used in their declarations.

3.5 Kinds

The equational logic underlying Maude is membership equational logic [106, 17]. In this logic sorts are grouped into equivalence classes called *kinds*. For this purpose, two sorts are grouped together in the same equivalence class if and only if they belong to the same connected component. Maude sorts are user-defined, while kinds are implicitly associated with connected components of sorts and are considered as "error supersorts." Terms (see Section 3.8) that have a kind but not a sort are understood as *undefined* or *error* terms.

In Maude modules, kinds are not independently and explicitly named. Instead, a kind is identified with its equivalence class of sorts and can be named by enclosing the name of one or more of these sorts in square brackets `[...]`; when using more than one sort, they are separated by commas.

For example, suppose we add a partial predecessor function to our `NUMBERS` module,

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

Then Maude will parse the term `p(zero)` and assign it the kind `[Nat]`, or equivalently `[NatSeq]` or also `[Nat, NatSeq]`, since the sorts `Nat` and `NatSeq` belong to the same connected component. Although any sort, or list of sorts in the connected component, can be enclosed in brackets to denote the corresponding kind, Maude uses a canonical representation for kinds; specifically, Maude prints the kind using a comma-separated list of the *maximal elements* of the connected component.

The Maude system also lifts automatically to kinds all the operators involving sorts of the corresponding connected components to form *error expressions*. Such error expressions allow us to give expressions to be evaluated the *benefit of the doubt*: if, when they are simplified, they have a legal sort, then they are okay; otherwise, the fully simplified error expression is returned, which the user can interpret as an error message. Equational simplification can also occur at the kind level, so that operators can map error terms to defined terms, which may be useful for error recovery.

It is also possible to explicitly declare operators at the kind level. This corresponds to declaring a partial operation, which is defined for those argument values for which Maude can determine that the resulting term has a sort. Note that the operation is considered to be total at the kind level. As an example, consider the following fragment of a graph specification:

```
sorts Node Edge .
ops source target : Edge -> Node .
sort Path .
subsort Edge < Path .
op _;_ : [Path] [Path] -> [Path] .
```

The sorts `Node` and `Edge`, along with the `source` and `target` operators mapping edges to nodes, axiomatize the basic graph concepts. The sort `Path` is intended to be the paths through the graph, sequences of edges with the target of one edge being the source of the next edge. Edges are singleton paths, and `_;_` denotes the *partial* concatenation operation, indicated by giving kinds rather than sorts in the argument list. Later, in Section 4.3, we will see how to specify when a sequence of edges has sort `Path`.

To emphasize the fact that an operator defined at the kind level in general defines only a *partial* function at the sort level, Maude also supports a notational variant in which an (always total) operator at the kind level can equivalently be defined as a partial operator between sorts in the corresponding kinds, with syntax '`~>`' instead of '`->`' to indicate partiality. For example, the above operator declaration can be equivalently specified by

```
op _;_ : Path Path ~> Path .
```

More generally, the partial operator declaration

`op` ⟨*OpName*⟩ `:` ⟨*Sort-1*⟩ `...` ⟨*Sort-k*⟩ `~>` ⟨*Sort*⟩ `.`

is equivalent to the total operator declaration at the kind level

`op` ⟨*OpName*⟩ `:` `[`⟨*Sort-1*⟩`]` `...` `[`⟨*Sort-k*⟩`]` `->` `[`⟨*Sort*⟩`]` `.`

3.6 Operator overloading

Operators in Maude can be *overloaded*, that is, we can have several operator declarations for the same operator with different arities and coarities. Consider extending our number module with a new sort `Nat3` (of natural numbers modulo 3), constants `0`, `1`, and `2` of sort `Nat3`, and two further operator declarations for `_+_`.

```
op _+_ : NzNat Nat -> NzNat .
sort Nat3 .
ops 0 1 2 : -> Nat3 .
op _+_ : Nat3 Nat3 -> Nat3 .
```

Now `_+_` is overloaded, having three declarations. However, there are two different kinds of overloading present in the example. The additional declaration of `_+_` with first argument `NzNat` is an example of *subsort overloading*. Here the two `_+_` operators on `Nat` and `NzNat` are supposed to have the same behavior on their shared argument values, that is, the operator on the subsort `NzNat` is the *restriction* of the operator on the larger sort `Nat`. The main point of such declarations is to give more sort information, for example that the result of adding a nonzero natural number to any natural number is nonzero. Many more examples of this form of overloading can be found in the predefined data modules for the number hierarchy (Chapter 7) and in other modules throughout the manual.

In contrast, the sorts `Nat` and `NzNat` on the one hand, and the sort `Nat3` on the other belong to two different *connected components* in the subsort ordering and therefore natural number addition and addition modulo 3 are semantically unrelated. This form of overloading is called *ad-hoc overloading*. Both subsort and ad-hoc overloading of operators are allowed in Maude. However, to avoid ambiguous expressions we require that if the sorts in the arities of two operators with the same syntactic form are pairwise in the same connected components, then the sorts in the coarities must likewise be in the same connected component.

Strictly speaking, this requirement would rule out ad-hoc overloaded constants. For this reason, we have declared two different constants `zero` and `0` for the corresponding zero elements. However, this requirement can be relaxed, and it is often natural to do so. For example, the constants of a parameterized module (see Chapter 6.3) can appear in many different connected components for different instances of the module, and it may be cumbersome to rename them all. To allow this relaxation, constants—and, more generally, terms (see Section 3.8)—can be *qualified by their sort*, by enclosing them in parentheses followed by a dot and the sort name. In this way, we could have instead declared `0` as an ad-hoc overloaded constant for natural numbers and for natural numbers modulo 3, and could then disambiguate the expression `0 + 0` by writing, for example, `0 + (0).Nat` and `0 + (0).Nat3`, or `(0 + 0).Nat` and `(0 + 0).Nat3`.

3.7 Variables

A variable is constrained to range over a particular sort or kind. Variables can be declared on-the-fly in Maude with syntax consisting of an identifier (the variable name), a colon, and

another identifier (its sort) or kind expression (its kind). For example, `N:Nat` declares a variable named `N` of sort `Nat`, and `X:[Nat]` declares a variable named `X` of kind `[Nat]`.

The scope of an on-the-fly variable declaration is the declaration's occurrence. Thus *each* such variable must be accompanied by its sort or kind.

A variable can also be declared in a module using the keyword `var` followed by an identifier (the variable name), followed by a colon with white space before and after, followed by an identifier (its sort) or kind expression (its kind), followed by white space and a period.

```
var N : Nat .
var X : [Nat] .
```

The scope of such a declaration is the entire module. It has the effect of replacing occurrences of `N` and `X` by the on-the-fly versions `N:Nat` and `X:[Nat]`.

Multiple variables of the same sort can be declared using the keyword `vars`.

```
vars M N : Nat .
vars X Y : [Nat] .
```

Both upper and lower case names for variables are possible. However, upper case variable names are more customary in Maude. The syntactic conventions for the acceptable names of variables in variable declarations are the same as those for constant operators, that is, for operators with empty arity. In particular, the underscore '_' cannot be used in the name of a variable, but the colon ':' can; thus the scanning for ':' in order to extract the appropriate sort or kind from an on-the-fly variable declaration is done from right to left.

3.8 Terms and preregularity

A term is either a constant, a variable, or the application of an operator to a list of argument terms. The sort of a constant or variable is its declared sort. In the application of an operator, the argument list must agree with the declared arity of the operator. That is, it must be of the same length, and each term must have sort (or at least kind) in the connected component of the corresponding declared argument sort. Using prefix form—which can always be used for *any* operator, regardless of having been declared with either prefix or mixfix syntax—the syntax of operator application is the operator's name followed by '`(`', followed by a list of argument terms separated by commas, followed by '`)`'. Here are some examples of prefix notation from our numbers module.

```
s_(zero)
s_(sd(N:Nat, M:Nat))
p(s_(zero))
_+_(N:Nat, M:Nat)
```

The application of an operator declared with mixfix form also has a mixfix syntax: the operator's mixfix name with each underscore replaced by the corresponding term from the argument list. The mixfix form of the above examples is

```
s zero
s sd(N:Nat, M:Nat)
p(s zero)
N:Nat + M:Nat
```

The *kind* of a term is the result kind of its topmost operator. For example, the kind of `p(s zero)` is `[Nat]`, since `Nat` is the result sort of `p`. If a module's grammar is unambiguous (see the discussion on parsing in the following section), then each term has a single kind. But

we can also associate *sorts* to terms. In general, even if the grammar is unambiguous, a term may have several sorts, due to the subsort ordering. Specifically, constants have the sort they are declared with and any supersort of it. Given a term of the form $f(t_1, \ldots, t_n)$, if t_i has sort s_i for $i = 1, \ldots, n$ and there is an operator declaration $f : s_1 \ldots s_n \to s$, then the term $f(t_1, \ldots, t_n)$ has sort s and any of its supersorts. For example, in our example `NUMBERS` module the term `s s 0` has sorts `NzNat` and `Nat`.

A very desirable property of a module is that each term has a *least* sort that can be assigned to it. Such a least sort gives us the most detailed information on how to classify such a term as a data element. For example, the least sort of the term `s s 0` is `NzNat`, and this gives us the most precise classification of such a term in the sort hierarchy. Given an arbitrary signature Σ, we can have terms that fail to have a least sort. However, if Σ satisfies a simple syntactic property called *preregularity* [78], we can guarantee that any Σ-term will have a least sort. We call Σ *preregular* if for each n, given an n-argument function symbol f and sorts $s_1, \ldots, s_n$ such that $f(x_1 : s_1, \ldots, x_n : s_n)$ is a well-formed Σ-term, then there is a least sort s among all the sorts s' appearing in (possibly overloaded) operator declarations of the form $f : s'_1, \ldots, s'_n \longrightarrow s'$ in Σ such that for $1 \leq i \leq n$ we have $s_i \leq s'_i$. For example, the signature

```
sorts A B C D .
subsorts A < B C < D .
op a : -> A .
op f : B -> B .
op f : C -> C .
```

fails to be preregular, because for the sort `A` the term `f(X:A)` is a well-formed term, but there is no least sort for the result of `f` with arguments greater or equal to `A`, since either `B` or `C` can be chosen as result sorts, and they are incomparable in the sort hierarchy. As a consequence, both `f(X:A)` and `f(a)` do not have a least sort: they have sorts `B`, `C`, and `D`, and `B` and `C` are minimal sorts among those sorts.

As already mentioned in Section 3.4 for the `assoc` attribute and further explained in Section 4.4.1, operators can be declared with equational axioms such as associativity (`assoc`), commutativity (`comm`), and identity (`id:`). This means that, if we denote by A the corresponding associativity and/or commutativity, and/or identity equations, we are not really interested in syntactic terms t, but rather in equivalence classes modulo A, that is, in the equivalence class $[t]_A$ of each term t, since all representatives of the class are viewed as equivalent representations. Preregularity *modulo* A now means that we can assign a least sort not just to any well-formed term t, but also to its equivalence class $[t]_A$. As further explained in Section 20.3.5, Maude assumes that modules are preregular modulo whatever axioms such as `assoc`, `comm`, and `id:` have been declared for operators, checks syntactic conditions ensuring preregularity modulo A, and generates warnings when a module fails to satisfy such preregularity conditions.

A *ground term* is a term containing no variables: only constants and operators. Intuitively, ground terms denote either data in case no equations apply to the term (for example, `s zero` is data) or functional expressions indicating how an equationally defined function is applied to data (for example, `(s zero) + (s zero)`). Ground terms modulo equations constitute the initial algebra associated with a specification, as discussed later in Section 4.3.

3.9 Parsing

As seen in previous sections, the Maude language supports user-definable syntax including mixfix operator declarations. Parsing is done in stages. First Maude's surface syntax is parsed using a bison/flex-based parser. Operator declarations are extracted and are used to construct

a grammar for language constructs that can contain terms. Finally those constructs are parsed using a version of Leo's algorithm for context-free grammars [91], which has been extended to handle Maude-specific features including the precedence-gather mechanism and bubbles.

With mixfix syntax, the occurrence of ambiguities in the parsing of terms is very common. Of course, we can always provide unambiguous grammars, which are frequently surprisingly large, or use parentheses for breaking the possible ambiguities. But usually we would like to have a more powerful alternative. Maude reduces such ambiguities by using a mechanism based on *precedence values* and *gathering patterns*.

Let us assume the following declarations for some arithmetic expressions:

```
sort Nat .
ops 1 2 3 : -> Nat .
ops _+_ _*_ : Nat Nat -> Nat .
```

An expression like `1 + 2 * 3` is ambiguous, since both `(1 + 2) * 3` and `1 + (2 * 3)` are valid parses. This kind of ambiguity is usually solved by assigning a *precedence* to each of the operators. In Maude, the precedence of an operator is given by a natural number,[3] where a lower value indicates a tighter binding.

Operator precedence then defines how an expression should be parsed when several operators are present. We can assign a precedence to an operator with a `precedence` (abbreviated `prec`) attribute, which takes the precedence value as an argument. For example, one would expect multiplication to be evaluated before addition. Thus, we can give precedences, e.g., 33 and 31 to the operators `_+_` and `_*_`, respectively, as follows:

```
op _+_ : Nat Nat -> Nat [prec 33] .
op _*_ : Nat Nat -> Nat [prec 31] .
```

The term `1 + 2 * 3` is now unambiguous: its only possible parse is `1 + (2 * 3)`.

Precedence can be overridden using parentheses; we can always write `(1 + 2) * 3` in case this is the term we are interested in. For those operators for which the user does not specify a precedence value, a default one is given (see Section 3.9.1 for a discussion on the default precedence values). For example, both operators `_+_` and `_*_` above get 41 as their default precedence, and hence the ambiguity.

The precedence mechanism is not enough, however. For example, the expression `1 + 2 + 3` is still ambiguous, because both parses `(1 + 2) + 3` and `1 + (2 + 3)` are possible. Usually, programming languages define a way of associating operators to solve this kind of problems, so that the *associativity* of the operators determines which is evaluated first. For example, addition usually is left-associative, and therefore we expect to parse it as `(1 + 2) + 3`. In Maude, we can specify not only the associativity of operators, but general *gathering patterns* for each operator.

The gathering pattern of an operator restricts the precedences of terms that are allowed as arguments. We give a (non-empty) sequence of as many `E`, `e`, or `&` values as the number of arguments in the operator, that is, one of these values for each argument position:

- `E` indicates that the argument must have a precedence value lower or equal than the precedence value of the operator,
- `e` indicates that the argument must have a precedence value strictly lower than the precedence value of the operator, and
- `&` indicates that the operator allows any precedence value for the corresponding argument.

[3] The maximum allowed precedence value is $2^{31} - 1$.

In fact, the precedence values work because of their combination with the gathering patterns. For example, the precedence values given to `_+_` and `_*_` work as expected because their *default* gathering pattern is `(E E)` (see Section 3.9.2), which forces them to be applied only to terms of smaller or equal precedence value. Thus, `1 + (2 * 3)` is a valid parse for `1 + 2 * 3`. On the other hand, since the precedence of a term is given by the precedence of its top operator, `(1 + 2) * 3` is not a valid parse for `1 + 2 * 3`, because the term `1 + 2` has precedence value 33, which is greater than the precedence of `_*_`.

Moreover, by default, all constants have precedence 0 (see Section 3.9.1), and therefore they are also valid arguments for both operators.

We can specify `_+_` and `_*_` as left-associative by giving to them gathering pattern `(E e)`.

```
op _+_ : Nat Nat -> Nat [prec 33 gather (E e)] .
op _*_ : Nat Nat -> Nat [prec 31 gather (E e)] .
```

In this way, we force the second argument of these operators to be of a strictly lower precedence. Then, a term with `_+_` as top operator (or any other operator with the same precedence) like `2 + 3` is nonvalid as second argument for `_+_`. But it would be valid as first argument, since terms with equal precedence are allowed. Now the only possible parse for the expression `1 + 2 + 3` is `(1 + 2) + 3`.

Note that parentheses could be described as an operator `(_)` with precedence 0 and gathering pattern `(&)`. Thus, any term can appear inside parentheses, and any subterm of a term can be enclosed in parentheses.

3.9.1 Default precedence values

Maude associates default precedence values to those operators for which the user does not specify this information as part of the operator declaration. The default precedence values are entirely similar to those used by OBJ3 [79]. The rules for the assignment of default precedence values are:

- Operators with standard form (constants and prefix operators) always have precedence 0, regardless of user settings. The user cannot change the precedence value or gathering pattern for operators in standard form.

- Mixfix operators which begin and end with something different from an underbar have precedence 0. Operators as, for example, `(_)`, `<_:_|_>`, and `if_then_else_fi` follow this rule.

- Mixfix operators which begin or end with an underbar have precedence 15 for a unary operator and 41 for everything else. Note that this 'or' is exclusive. Operators like, e.g., `not_`, `_!`, or `to_:_` fall into this category.

- Mixfix operators which begin and end with an underbar have precedence 41. This rule applies, e.g., to the operators `__`, `_+_`, `_*_`, and `_?_:_`.

3.9.2 Default gathering patterns

As for precedence values, Maude assigns default gathering patterns to all those operators for which the user does not specify this information as part of the operator declaration. The default gathering patterns are also entirely similar to those used by OBJ3 [79]. The rules for the assignment of the default gathering patterns are:

- All arguments of prefix operators have a gathering value `&`, regardless of the user specification.

- If the underbar corresponding to an argument is not adjacent to another underbar, and it is neither the leftmost nor the rightmost token in the operator, then the default gathering value for such an argument is `&`. In other words, if an underbar appears between tokens different from the underbar, then its corresponding argument will have this default gathering pattern. For example, the default gathering pattern for the operator `if_then_else_fi` is `(& & &)`, the default gathering pattern for the operator `[_and then_]` is `(& &)`, and the default gathering pattern for the operator `(_)` is `(&)`.

- If the underbar corresponding to an argument is adjacent to another underbar, or if it is the leftmost or the rightmost token in the operator, then the default gathering value for such an argument is `E`. Thus, e.g., the default gathering pattern for the operator `not_` is `(E)`, the default gathering pattern for the operator `_?_:_` is `(E & E)`, the default gathering pattern for the operator `_+_` is `(E E)`, and the default gathering pattern for the operator `__` is `(E E)`.

  Those binary operators which start with an underscore, end with an underscore, and have a precedence greater than 0 are handled as special cases:

  – The operator will have gathering pattern `(e E)` if it has the `assoc` attribute (see Section 4.4.1). For example, the following operators fall into this category.

  ```
  op _+_ : Nat Nat -> Nat [assoc] .
  op _*_ : Nat Nat -> Nat [assoc] .
  op __ : NatList NatList -> NatList [assoc] .
  ```

  – If the operator does not have the `assoc` attribute, but its first argument, its last argument, and its coarity are in the same connected component of sorts, then:

    1. if the subsort relations allow it to right-associate but not left-associate, then the first argument's gathering pattern will change to `e`, and
    2. if the subsort relations allow it to left-associate but not right-associate, then the last argument's gathering pattern will change to `e`.

  Assuming `Int < IntList`, then the operators

  ```
  op _<:_ : Int IntList -> IntList .
  op _:>_ : IntList Int -> IntList .
  ```

  have, by default, gathering patterns `(e E)` and `(E e)`, respectively. According to the general rule, since their argument bars are the leftmost and the rightmost tokens, the gathering pattern should be `(E E)` for both of them. However, both operators fall into the second special case, since they are binary operators which start and end with underscores, have a precedence greater than 0 (by default 41), and are not declared associative. Given the subsort relation, the operator `_<:_` may right-associate, but not left-associate, that is, `1 <: 2 <: 3` should be parsed as `1 <: (2 <: 3)`, but `(1 <: 2) <: 3` should not be a valid parse. Therefore, `_<:_` gets default gathering pattern `(e E)`. And similarly for `_:>_`, although in this case it can left-associate, and therefore it gets default gathering pattern `(E e)`.

3.9.3 The extended signature of a module

In addition to the signature defined by the user, parsing of terms takes place in an extended grammar in which information for handling parentheses, sort and equality predicates, `if_then_else_fi`, and qualification operators are included. These structures belong to the so-called *extended signature of a module*. The main structures added in the extended signature of a module are:

- *Sort disambiguation.* For each sort `S` in the signature of a module, Maude adds to the signature the operator

  ```
  op (_).S : S -> S .
  ```

  This helps in the disambiguation of ad-hoc overloaded constants and terms. As an example, remember from Section 3.6 that if we declare `0` as an ad-hoc overloaded constant for natural numbers and for natural numbers modulo 3, then we can disambiguate the expression `0 + 0` by writing, for example, `0 + (0).Nat` and `0 + (0).Nat3`, or `(0 + 0).Nat` and `(0 + 0).Nat3`. As another example, in the module `META-MODULE` (see Section 17.4), the term `none` is ambiguous, since the operator `none` is used as the empty set of operator declarations, equations, rules, etc. We can disambiguate it by writing `(none).OpDeclSet`. Of course, these disambiguation operators can be used not only for constants, but for any term. For example, we can write `(2 + 3).Nat` as a valid term in the predefined module `NAT`.

- *Parentheses.* The extended signature of a module contains the operator

  ```
  op (_) : S -> S .
  ```

  for each sort `S` in its signature. These operators allow the use of parentheses without having to declare a parentheses operator for each sort. For example, `(2 + 3)`, `(2 + 3) + 5`, `(2 + (3) + 5)`, `(((2 + 3)) + 5)`, are all valid terms in `NAT`, thanks to these declarations.

- *Equivalent single-identifier form* for all operators. Each declared operator, including those in mixfix form, may also be used in their equivalent single-identifier prefix form. For example, in the `NAT` module, the term `_+_(2, 3)` is equivalent to `2 + 3`, and the terms `if true then 2 + 3 else - 3 fi` and `if_then_else_fi(true, _+_(2, 3), -_(3))` are equivalent; any combination is possible so `if_then_else_fi(true, 2 + 3, - 3)` is also valid.

- *Flattened associative argument lists.* Operators with the attribute `assoc` may be used in Maude in a nonparenthesized flattened form (see Section 4.8). This is possible thanks to the precedence-gathering values in mixfix notation, but it is also possible in prefix syntax. For example, `gcd(2, 3, 4)` is a valid term in `NAT`, where `gcd` is the greater common divisor operator, which is declared as a binary associative operator. Of course, this term can always be written in the standard format as `gcd(2, gcd(3, 4))` or `gcd(gcd(2, 3), 4)`. Furthermore, we can combine this possibility with the single-identifier form to write things like `_+_(2, 3, 4)` instead of `_+_(_+_(2, 3), 4)` or `_+_(2, _+_(3, 4))`, but of course, since `_+_` is declared with the `assoc` attribute in the predefined module `NAT`, we can just write `2 + 3 + 4`.

- *Polymorphic operators and the* `BOOL` *module.* All the information contained in the predefined modules `TRUTH-VALUE`, `TRUTH`, `BOOL-OPS`, and `BOOL` is included in the extended signature of each module (unless this inclusion is explicitly disabled). In particular, appropriate

instances of the polymorphic operators contained in `TRUTH` (that is, `if_then_else_fi`, `_==_`, and `_=/=_`) are generated for each sort in the module; in addition, for each sort `S`, a sort predicate `_:: S` is also added. All these modules and operators are fully explained in Section 7.1.

3.9.4 Parsing examples

Maude provides the `parse` command for parsing terms. The command does not do anything other than parsing the given term in the extended signature of the module. This is exactly what is done when a term appears in a command, before executing such a command. For example, when we try to reduce a term `(2 + 3) * 5`, the system first parses it and then reduces it. If the term is ambiguous, or there is no parse for it, an error message is given and no further action takes place.

```
Maude> reduce in NAT : 2 + true .
Warning: <standard input>, line 1:
  didn't expect token true: 2 + true <---*HERE*
Warning: <standard input>, line 1: no parse for term.
```

For testing the parsing of terms we can use the `parse` command.

```
Maude> parse in NAT : 2 + true .
Warning: <standard input>, line 1:
  didn't expect token true: 2 + true <---*HERE*
Warning: <standard input>, line 1: no parse for term.
```

As other commands, parsing can take place either in the module explicitly mentioned in the command or in the current module.

We illustrate the use of the `parse` command for the examples introduced in the previous sections. Let us first consider a module `PARSING-EX1` with constants `1`, `2`, and `3`, and binary operators `_+_` and `_*_`.

```
fmod PARSING-EX1 is
  sort Nat .
  ops 1 2 3 : -> Nat .
  ops _+_ _*_ : Nat Nat -> Nat .
endfm
```

Since `_+_` and `_*_` are declared without precedence values, and therefore both get the default value 41, we obtain the following result.

```
Maude> parse 1 + 2 * 3 .
Warning: <standard input>, line 13: ambiguous term, two parses are:
1 + (2 * 3) -versus- (1 + 2) * 3

Arbitrarily taking the first as correct. Nat: 1 + (2 * 3)
```

As a first solution, we may consider using parentheses.

```
Maude> parse in PARSING-EX1 : 1 + (2 * 3) .
Nat: 1 + (2 * 3)

Maude> parse in PARSING-EX1 : (1 + 2) * 3 .
Nat: (1 + 2) * 3
```

Let us now consider the module `PARSING-EX2`, where `_+_` and `_*_` are declared with precedences 33 and 31, respectively.

```
fmod PARSING-EX2 is
  sort Nat .
  ops 1 2 3 : -> Nat .
  op _+_ : Nat Nat -> Nat [prec 33] .
  op _*_ : Nat Nat -> Nat [prec 31] .
endfm
```

Now, parentheses are not necessary for parsing the term `1 + 2 * 3`.

```
Maude> parse in PARSING-EX2 : 1 + 2 * 3 .
Nat: 1 + 2 * 3
```

Of course, we may still use parentheses.

```
Maude> parse in PARSING-EX2 : (1 + 2) * 3 .
Nat: (1 + 2) * 3
```

Since the default gathering patterns for binary operators like `_+_` and `_*_` is `(E E)`, a term like `1 + 2 + 3` is ambiguous.

```
Maude> parse in PARSING-EX2 : 1 + 2 + 3 .
Warning: <standard input>, line 30: ambiguous term, two parses are:
1 + (2 + 3) -versus- (1 + 2) + 3

Arbitrarily taking the first as correct. Nat: 1 + (2 + 3)
```

As above, we may use parentheses to parse such terms.

```
Maude> parse in PARSING-EX2 : (1 + 2) + 3 .
Nat: (1 + 2) + 3

Maude> parse in PARSING-EX2 : 1 + (2 + 3) .
Nat: 1 + (2 + 3)
```

Let us now consider the module `PARSING-EX3`, where `_+_` and `_*_` are declared to be left-associative, that is, with gathering patterns `(E e)`.

```
fmod PARSING-EX3 is
  sort Nat .
  ops 1 2 3 : -> Nat .
  op _+_ : Nat Nat -> Nat [prec 33 gather (E e)] .
  op _*_ : Nat Nat -> Nat [prec 31 gather (E e)] .
endfm
```

Now, the terms above have unambiguous parses.

```
Maude> parse in PARSING-EX3 : 1 + 2 * 3 .
Nat: 1 + 2 * 3

Maude> parse in PARSING-EX3 : 1 + 2 + 3 .
Nat: 1 + 2 + 3
```

Let us now consider the module `PARSING-EX4`, where `_+_` and `_*_` are declared to be associative. Note that in this case, by default, they are assigned gathering patterns `(E e)`.

```
fmod PARSING-EX4 is
  sort Nat .
  ops 1 2 3 : -> Nat .
  op _+_ : Nat Nat -> Nat [prec 33 assoc] .
  op _*_ : Nat Nat -> Nat [prec 31 assoc] .
endfm
```

```
Maude> parse in PARSING-EX4 : 1 + 2 * 3 .
Nat: 1 + 2 * 3

Maude> parse in PARSING-EX4 : 1 + 2 + 3 .
Nat: 1 + 2 + 3
```

We illustrate the use of the extended signature in which all terms are parsed with the following examples.

```
Maude> parse in PARSING-EX1 : (2 + 3).Nat .
Nat: 2 + 3

Maude> parse in PARSING-EX1 : (2).Nat + 3 .
Nat: 2 + 3

Maude> parse in PARSING-EX1 : (2).Nat + (3).Nat .
Nat: 2 + 3

Maude> parse in PARSING-EX1 : ((1) + ((2) + (3))) .
Nat: 1 + (2 + 3)

Maude> parse in PARSING-EX1 : _+_(1, _+_(2, 3)) .
Nat: 1 + (2 + 3)

Maude> parse in PARSING-EX4 : _+_(1, 2, 3) .
Nat: 1 + 2 + 3

Maude> parse in PARSING-EX4 : if 1 == 2 then 1 + 2 else _+_(1, 2) fi .
Nat: if 1 == 2 then 1 + 2 else 1 + 2 fi

Maude> parse in PARSING-EX4 :
        if _==_(1, 2)
        then if_then_else_fi(1 + 2 :: Nat, 1 * 1, 2 * 1)
        else _+_(1, 2)
        fi .
Nat: if 1 == 2
     then if (1 + 2) :: Nat
          then 1 * 1
          else 2 * 1
          fi
     else 1 + 2
     fi
```

Chapter 4

Functional Modules

Functional modules define data types and operations on them by means of equational theories. The data types consist of elements that can be named by ground terms. Two ground terms denote the same element if and only if they belong to the same equivalence class as determined by the equations. That is, the mathematical semantics of a functional module is its *initial algebra*. Maude's functional modules are assumed to have the nice property that equations, considered as simplification rules by using them only in the left to right direction, are Church-Rosser and terminating (see Section 4.7). This means that repeated application of the equations as simplification rules eventually reaches a term to which no further equations apply, and the result, called the *canonical form*, is the same regardless of the order of application of the equations. Thus each equivalence class has a natural representative, its canonical form, that can be computed by equational simplification. As explained in Section 1.2, this ensures that the initial algebra and the canonical term algebra of the functional module are isomorphic, and therefore that the module's mathematical and operational semantics coincide.

The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic [78] called *membership equational logic* [106, 17]. Thus, functional modules support multiple sorts, subsort relations, operator overloading, and assertions of membership in a sort.

As was mentioned in Section 3.2, a functional module is declared in Maude using the keywords

`fmod` ⟨*ModuleName*⟩ `is` ⟨*DeclarationsAndStatements*⟩ `endfm`

For example,

```
fmod NUMBERS is
  ...
endfm
```

declares a module named `NUMBERS`. The dots stand for the actual declarations and statements that may appear in the functional module. Declarations include the importation of other functional modules (see Chapter 6), and sort, subsort, and operator declarations. Statements include equational and membership axioms. Declarations were discussed in Chapter 3. What remains to be explained are equational and membership statements.

4.1 Unconditional equations

Unconditional equations are declared using the keyword `eq`, followed by a term (its lefthand side), the equality sign `=`, then a term (its righthand side), optionally followed by a list of statement attributes (see Section 4.5 later in this chapter) enclosed in square brackets, and ending with white space and a period. Thus the general scheme is the following:

`eq` ⟨*Term-1*⟩ `=` ⟨*Term-2*⟩ `[`⟨*StatementAttributes*⟩`]` .

The terms `t` and `t'` in an equation `t = t'` must both have the same kind. In order for the equation to be executable, any variable appearing in `t'` must also appear in `t`. Equations not satisfying this requirement can also be declared (for example, to document a lemma holding true in the module) but in such a case they should always be specified with the `nonexec` attribute (see Section 4.5.3). We can add equations axiomatizing the addition operation in our `NUMBERS` module as follows, where we distinguish two cases for the second argument, according to whether it is zero or not:

```
vars N M : Nat .
eq N + zero = N .
eq N + s M = s (N + M) .
```

The following equations define the symmetric difference operation `sd` on natural numbers, which returns the result of subtracting the smaller from the larger of its two arguments.

```
eq sd(N, N) = zero .
eq sd(N, zero) = N .
eq sd(zero, N) = N .
eq sd(s N, s M) = sd(N, M) .
```

In general, in a functional module one can specify equations (and also conditional equations, as explained in Section 4.3) in three different ways:

1. in the style given above, in which case they are assumed to be executable as simplification rules from left to right;

2. in the same style as above, but with the `nonexec` attribute (see Section 4.5.3), in which case Maude does not use them for simplification (except at the metalevel with a user-given strategy, see Section 17.7); and

3. as equational attributes of specific operators (see Section 4.4.1).

For example, a binary operator `f` can be declared `assoc` and `comm`, telling Maude that it satisfies the associativity and commutativity axioms. Such equational attributes should *not* be written explicitly as equations in the specification. There are two reasons for this. Firstly, this is redundant, since they have already been declared as equational attributes. Secondly, although declaring such equations either only explicitly as equations, or twice—one time as equational attributes and another as explicit equations—does not affect the *mathematical semantics* of the specification, that is, the initial algebra that the specification denotes (see Section 4.3), it does however drastically alter the specification's *operational semantics*. For example, if the `comm` attribute for `f` were to be stated as an equation `f(X, Y) = f(Y, X)`, then using the equation as a simplification rule applied to the term, say, `f(a, b)`, would lead to the nonterminating chain of equational simplifications

```
f(a, b) = f(b, a) = f(a, b) = f(b, a) = ...
```

This is quite bad, since we want the equations specified by method (1) to be used as simplification rules and assume them to be terminating and Church-Rosser, so that they always simplify a term to a unique result that cannot be further simplified. Instead, if `comm` is declared as an equational attribute, the above kind of looping does not happen: Maude then simplifies terms *modulo* the declared equational attributes, so that the terms `f(a, b)` and `f(b, a)` would indeed be treated as identical. For more on equational attributes see Section 4.4.1.

4.2 Unconditional memberships

Unconditional membership axioms specify terms as having a given sort. They are declared with the keyword `mb` followed by a term, followed by '`:`', followed by a sort (that must always be in the same kind as that of the term), followed by a period. As equations, memberships can optionally have statement attributes (see Section 4.5).

`mb` ⟨*Term*⟩ `:` ⟨*Sort*⟩ `[`⟨*StatementAttributes*⟩`]` .

To illustrate this, consider the module `3*NAT` with the basic Peano number declarations as in the `NUMBERS` module and a new sort `3*Nat`.

The fact that `3*Nat` consists of multiples of 3 is expressed using the subsort declaration `Zero < 3*Nat < Nat` and the membership statement `mb (s s s M3) : 3*Nat` for `M3` a variable of sort `3*Nat`.

```
fmod 3*NAT is
  sort Zero Nat .
  subsort Zero < Nat .
  op zero : -> Zero .
  op s_ : Nat -> Nat .

  sort 3*Nat .
  subsorts Zero < 3*Nat < Nat .
  var M3 : 3*Nat .
  mb (s s s M3) : 3*Nat .
endfm
```

Memberships axioms can interact in undesirable ways with operators that are declared with the `assoc` or `iter` attributes (see later Sections 4.4.1 and 4.4.2, respectively). This is explained and illustrated with examples in Sections 20.3.8 and 20.3.9.

4.3 Conditional equations and memberships

Equational conditions in conditional equations and memberships are made up of individual equations $t = t'$ and memberships $t : s$. A condition can be either a single equation, a single membership, or a conjunction of equations and memberships using the binary conjunction connective `/\` which is assumed to be associative. Thus the general form of conditional equations and memberships is the following:

`ceq` ⟨*Term-1*⟩ `=` ⟨*Term-2*⟩
  `if` ⟨*EqCondition-1*⟩ `/\` ... `/\` ⟨*EqCondition-k*⟩
  `[`⟨*StatementAttributes*⟩`]` .

`cmb` ⟨*Term*⟩ `:` ⟨*Sort*⟩
  `if` ⟨*EqCondition-1*⟩ `/\` ... `/\` ⟨*EqCondition-k*⟩
  `[`⟨*StatementAttributes*⟩`]` .

Furthermore, the concrete syntax of equations in conditions has three variants, namely:

- ordinary equations `t = t'`,
- *matching equations* `t := t'`, and
- *abbreviated Boolean equations* of the form `t`, with `t` a term in the kind `[Bool]`, abbreviating the equation `t = true`.

Any term `t` in the kind `[Bool]` can be used as an abbreviated Boolean[1] equation. The Boolean terms appearing most often in abbreviated Boolean equations are terms using the built-in equality `_==_` and inequality `_=/=_` predicates, and the built-in membership predicates `_:: S` with `S` a sort, including Boolean combinations of such terms with `not_`, `_and_`, `_or_` and other Boolean connectives (see Section 7.1 for a detailed description of all these operators). For example, the following Boolean terms in the `NUMBERS` module (assuming that a "greater than" operator `_>_` has also been defined in `NUMBERS`),

```
N == zero
M =/= s zero
not (K :: NzNat)
(N > zero or M =/= s zero)
```

can appear as abbreviated Boolean equations in a condition, abbreviating, respectively, the equations:

```
(N == zero) = true
(M =/= s zero) = true
not (K :: NzNat) = true
(N > zero or M =/= s zero) = true
```

To illustrate the use of conditional equations and memberships, let us reconsider the path example from Section 3.5. The following conditional statements express the key membership defining path concatenation and the associativity of this operator:

```
var  E : Edge .
vars P Q R S : Path .
cmb E ; P : Path if target(E) = source(P) .
ceq (P ; Q) ; R = P ; (Q ; R)
  if target(P) = source(Q) /\ target(Q) = source(R) .
```

The conditional membership axiom (introduced by the keyword `cmb`) states that an edge concatenated with a path is also a path when the target node of the edge coincides with the source node of the path. This has the effect of defining path concatenation as a partial operation on paths, although it is total on the kind `[Path]` of "confused paths."

Assuming variables `P`, `E`, and `S` declared as above, `source` and `target` operations over paths are defined by means of conditional equations with matching equations in conditions as follows:[2]

```
ceq source(P) = source(E) if E ; S := P .
ceq target(P) = target(S) if E ; S := P .
```

Matching equations[3] are mathematically interpreted as ordinary equations; however, operationally they are treated in a special way and they must satisfy special requirements. Note that

[1] By default, any Maude module imports the predefined `BOOL` module (see Section 7.1).

[2] Note that the `source` and `target` operations can equivalently be declared as

```
eq source(E ; S) = source(E) .
eq target(E ; S) = target(S) .
```

[3] Similar constructs are used in languages like ASF+SDF [42] and ELAN [12].

the variables `E` and `S` in the above matching equations do not appear in the lefthand sides of the corresponding conditional equations. In the execution of these equations, these new variables become instantiated by *matching* the term `E ; S` against the canonical form of the subject term bound to the variable `P` (see Section 4.7). In order for this match to decide the equality with the ground term bound to `P`, the term `E ; S` must be a *pattern*. Given a functional module M, we call a term t an *M-pattern* if for any well-formed substitution σ such that for each variable x in its domain the term $\sigma(x)$ is in canonical form with respect to the equations in M, then $\sigma(t)$ is also in canonical form. A sufficient condition for t to be an *M-pattern* is the absence of unifiers between its nonvariable subterms and lefthand sides of equations in M.

Ordinary equations $t = t'$ in conditions have the usual operational interpretation, that is, for the given substitution σ, $\sigma(t)$ and $\sigma(t')$ are both reduced to canonical form and are compared for equality, modulo the equational attributes specified in the module's operator declarations such as associativity, commutativity, and identity. Finally, abbreviated Boolean equations are just a special case of ordinary equations once they are expanded out.

The satisfaction of the conditions is attempted sequentially from left to right. Since in Maude matching takes place modulo equational attributes, in general many different matches may have to be tried until a match of all the variables satisfying the condition is found.

The above equations for `source` and `target` illustrate the use of matching equations to bind variables locally, in much the same way that `let` is used in some functional programming languages. In this example, since the matching is purely syntactic, the matching substitution is unique and gives a simple way to name parts of a structure or to name a complicated expression which appears multiple times in the main equation.

For M-patterns where some operators are matched modulo some equational attributes, matching substitutions need not be unique. This provides another way of using matching equations, namely to perform a search through a structure without any need to explicitly define a function that does this. For example, for sequences of natural numbers we can define a predicate `_occurs-inner_` that determines if a number occurs in a sequence other than at one of the ends. If one only cares about positive results,[4] the following will work.

```
op _occurs-inner_ : [Nat] [NatSeq] -> [Bool] .
ceq N:Nat occurs-inner NS:NatSeq = true
  if (NS0:NatSeq N:Nat NS1:NatSeq) := NS:NatSeq .
```

Note that this equation could also be written as

```
eq N:Nat occurs-inner NS0:NatSeq N:Nat NS1:NatSeq = true .
```

In both cases we check whether the sequence contains the natural number `N:Nat`, but making sure that the sequence contains other elements both before and after `N:Nat`.[5] With the above definition added to the numbers module, the term

```
zero occurs-inner (zero zero zero zero zero)
```

reduces to `true`, while the term

[4] Note that, since when the predicate is not true it remains unevaluated, we have defined it at the kind level, that is, as a partial Boolean function; however, using the `owise` attribute (see Section 4.5.4) it is very easy to add an extra equation making `_occurs-inner_` a *total* Boolean function.

[5] Note that here we assume the declaration of the `NatSeq` concatenation operator `__` as given in page 34, where it is declared to be associative. If we consider the declaration of this operator given in page 54, which is also declared to have `nil` as identity element, then we should write this equation as

```
op _occurs-inner_:  [Nat] [NatSeq] -> [Bool] .
ceq N:Nat occurs-inner NS:NatSeq = true
  if (I:Nat NS0:NatSeq N:Nat NS1:NatSeq M:Nat) := NS:NatSeq .
```

since the variables `NS0:NatSeq` and `NS1:NatSeq` might be instantiated to `nil`.

```
zero occurs-inner (zero zero)
```

does not reduce further.

Matching equations in conditions give great expressive power, but some care is needed in using them to define operations. Consider adding the following to the numbers module, in an attempt to define a test for the presence of `s s zero` in a sequence of natural numbers.

```
op hasTwo : [NatSeq] -> [Bool] .
ceq hasTwo(NS:NatSeq) = N:Nat == s s zero
  if NS0:NatSeq N:Nat NS1:NatSeq := NS:NatSeq .
```

With this addition to the numbers module, `hasTwo(zero zero)` does not get reduced, since the condition requires at least three numbers in the sequence. The term `hasTwo(zero (s s zero) zero)` reduces to `true`. The term `hasTwo(zero (s zero) (s s zero) zero)` also gets reduced, although it may return `true` or `false`; probably not what was intended. The problem is that there are several matches, each giving a different answer, so the conditional equation does not define a function. In fact, this conditional equation causes the Church-Rosser property to fail, and semantically identifies `true` and `false`, thus leading to an inconsistent theory. In contrast, as will be seen in Chapter 5, a *rule* with such a matching condition is not a problem, and does have the effect of searching a sequence of natural numbers for `s s zero`.

In summary, all the sort, subsort, and operator declarations and all the statements in a functional module (plus the functional modules imported if any) define an *equational theory* in *membership equational logic* [106, 17]. Such a theory can be described in mathematical notation as a pair $(\Sigma, E \cup A)$, where Σ is the *signature*, that is, the specification of the sorts, subsorts, kinds, and operators in the module, E is the collection of statements (equations and memberships, possibly conditional) and A is the set of equational attributes, such as `assoc` and `comm`, declared for some operators (that is, extra equations that are treated in a special way by the Maude interpreter to simplify modulo such attributes, see Section 4.4.1).

The family of ground terms definable in the syntax of Σ defines a model called a Σ-algebra and denoted T_Σ. In T_Σ, terms syntactically different denote different elements, so that T_Σ will *not* satisfy the equations in $E \cup A$, unless they are trivial equations such as $f(X) = f(X)$. The question is, what is the *optimal model* of the theory $(\Sigma, E \cup A)$? Goguen and Burstall's answer is: a model satisfying the axioms $E \cup A$ and such that it has *no junk* (that is, all elements can be denoted by ground Σ-terms), and *no confusion* (that is, only elements that are *forced to be equal* by the axioms $E \cup A$ are identified). Such a model, called the *initial algebra* of the equational theory $(\Sigma, E \cup A)$, exists [106], is denoted $T_{\Sigma/E \cup A}$, and provides the *mathematical semantics* of the Maude functional module specifying $(\Sigma, E \cup A)$.

Mathematically, $T_{\Sigma/E \cup A}$ can be constructed as the quotient of T_Σ in which the equivalence classes are those terms that are *provably equal* using the axioms $E \cup A$. Operationally, assuming that the axioms E are Church-Rosser and terminating modulo A (see Section 4.7), there is a much more intuitive equivalent description of $T_{\Sigma/E \cup A}$, namely as the family of *canonical forms* for the ground Σ-terms modulo A, that is, those terms that cannot be further simplified by the equations in E modulo A. That is, as explained in Section 1.2, we have then an isomorphism

$$T_{\Sigma/E \cup A} \cong Can_{\Sigma/E \cup A}$$

between the initial algebra $T_{\Sigma/E \cup A}$ and the canonical term algebra $Can_{\Sigma/E \cup A}$.

The Maude interpreter computes such canonical forms, which can be viewed as the *values* denoted by the corresponding functional expressions, with the `reduce` command (see Section 23.2 for details and Section 4.9 for examples).

4.4 Operator attributes

Operator declarations may include attributes that provide additional information about the operator: semantic, syntactic, pragmatic, etc. All such attributes are declared within a single pair of enclosing square brackets, '`[`' and '`]`', after the sort of the result and before the ending period. We discuss each of the categories of operator attributes below.

4.4.1 Equational attributes

Equational attributes are a means of declaring certain kinds of equational axioms in a way that allows Maude to use these equations efficiently in a built-in way. Currently Maude supports the following equational attributes:

- `assoc` (associativity),
- `comm` (commutativity),
- `idem` (idempotency),
- `id:` ⟨*Term*⟩ (identity, with the corresponding term for the identity element),
- `left id:` ⟨*Term*⟩ (left identity, with the corresponding term for the left identity element), and
- `right id:` ⟨*Term*⟩ (right identity, with the corresponding term for the right identity element).

An operator can be declared with several of these attributes, which may appear in any order in the attribute declaration. However, these attributes are only allowed for *binary* operators satisfying the following requirements:

- For `left id:`, it is required that the right domain sort and the range sort belong to the same kind.
- For `right id:`, it is required that the left domain sort and range sort belong to the same kind.
- For `assoc`, `comm`, `id:`, and `idem`, both domain sorts and the range sort must belong to the same kind.

These requirements are checked at parse time, and if the check fails a warning is output and the operator loses its attributes.

Furthermore, we have the following additional requirements:

- The attribute `idem` cannot be used in any combination of attributes that includes `assoc`, because the necessary matching and normalization algorithms have not been implemented yet. This requirement is quietly enforced by ignoring the attribute `idem` where necessary.
- Only one identity attribute (`left id:`, `right id:`, or `id:`) is allowed. This is enforced by a warning and by ignoring all but the first such attribute.
- Combining the attribute `comm` with either `left id:` or `right id:` silently turns the identity attribute into an `id:`.

- All subsort-overloaded instances of an operator must have the same attributes. This is further explained in Section 4.4.6.

Semantically, declaring a set of equational attributes for an operator is equivalent to declaring the corresponding equations for the operator. Operationally, using equational attributes to declare such equations avoids termination problems and leads to much more efficient evaluation of terms containing such an operator. In fact, the effect of declaring equational attributes is to compute with equivalence classes modulo such equations. This, besides being very expressive, avoids what otherwise would be insoluble termination problems. For example, if a commutativity equation like `x + y = y + x` is declared as an ordinary equation, then it will easily produce looping, nonterminating simplifications. If it is instead declared with an equational attribute `comm`, this looping behavior does not happen.

In our numbers example we can add a constant `nil` for the empty sequence and refine the declaration of sequence concatenation so that concatenation is associative with identity `nil`.

```
op nil : -> NatSeq .
op __ : NatSeq NatSeq -> NatSeq [assoc id: nil] .
```

As another example, we can form lists of Booleans as a supersort `BList` of `Bool` in an extension of the `BOOL` module (see Section 7.1) with a "cons" operator `_._` having `nil` as a right identity:

```
sort BList .
subsort Bool < BList .
op nil : -> BList .
op _._ : Bool BList -> BList [right id: nil] .
```

Note that, when equational attributes are declared, equational simplification using the other equations in the module does not take place at the purely syntactic level of replacing syntactic equals by equals, but is understood *modulo* the equational attributes. Therefore, the proper understanding of the notions of Church-Rosser and terminating equations, and of canonical forms, is now *modulo* the equational attributes that have been declared. We discuss matching and equational simplification modulo axioms in Section 4.8.

For example, by declaring the addition operation on natural numbers modulo 3 as commutative,

```
op _+_ : Nat3 Nat3 -> Nat3 [comm] .
```

it is enough to have the following equations to define its behavior on all possible combinations of arguments:

```
vars N3 : Nat3 .
eq N3 + 0 = N3 .
eq 1 + 1 = 2 .
eq 1 + 2 = 0 .
eq 2 + 2 = 1 .
```

The equations

```
eq 0 + N3 = N3 .
eq 2 + 1 = 0 .
```

are not needed, because they are subsumed by the first and third equations above, due to commutativity of `_+_`.

Notice that membership axioms and matching modulo associativity can interact in undesirable ways, as explained in Section 20.3.8.

4.4.2 The `iter` attribute

Maude provides a built-in mechanism called the `iter` (short for *iterated* operator) theory whose goal is to permit the efficient input, output, and manipulation of very large stacks of a unary operator.

Unary operators may be declared to belong to the `iter` theory by including `iter` in their attributes. After declaring

```
sort Foo .
op f : Foo -> Foo [iter] .
```

the term `f(f(f(X:Foo)))` can be input as `f^3(X:Foo)` and will be output in that form. A term such as `f^12345678901234567890(X:Foo)` is too large to be input, output or manipulated in regular notation, but can be input and output in this compact notation and certain (built-in) manipulations may thus be made efficient.

The precise form of the compact iter theory notation is the prefix name of the operator followed by `^[1-9][0-9]*` (in Lex regular expression notation) with no intervening white space. Note that `f^0123(X:Foo)` is not acceptable. Of course, regular notation (and mixfix notation if appropriate) can still be used.

Membership axioms may also interact in undesirable ways with operators declared with the `iter` attribute; see Section 20.3.9 for details.

4.4.3 Constructors

Assuming that the equations in a functional module are (ground) Church-Rosser and terminating, then every ground term in the module (that is, every term without variables) will be simplified to a canonical form, perhaps modulo some declared equational attributes. *Constructors* are the operators appearing in such canonical forms. The operators that "disappear" after equational simplification are instead called *defined functions*. For example, typical constructors in a sort `Nat` are `zero` and `s_`, whereas in the sort `Bool`, `true` and `false` are the only constructors.

It is quite useful for different purposes, including both debugging (see Chapter 20) and theorem proving, to specify when a given operator is a constructor. This can be done with the `ctor` attribute. For example, we can refine our operator declarations in Section 3.4 with constructor information as follows:

```
op zero : -> Zero [ctor] .
op s_ : Nat -> NzNat [ctor] .
op nil : -> NatSeq [ctor].
op __ : NatSeq NatSeq -> NatSeq [ctor assoc id: nil] .
```

Three slightly subtle points should be mentioned, namely the relationships of constructors to operator overloading, to kinds, and to equations. The first key observation is that constructor declarations are *local to given sorts for the arguments and for the result*. Nothing prevents an operator from being a constructor at some level in the subsort ordering but being a defined function at another. For example, we could have declared a successor function for integers,

```
op s_ : Int -> Int .
```

which is *not* a constructor. Indeed, we can define the sort `Int` with a subsort `NzNeg` of nonzero negative numbers built up with a unary minus constructor `-_`, and we can then specify both unary minus `-_` and successor `s_` as *defined functions* on the integers by giving the equations:

```
sorts NzNeg Int .
subsorts Nat NzNeg < Int .
```

```
op -_ : NzNat -> NzNeg [ctor] .
op -_ : Int -> Int .
op s_ : Int -> Int .

var N : Nat .

eq - zero = zero .
eq - (- (s N)) = s N .
eq s (- (s N)) = - N .
```

A related observation is that a defined function, which totally disappears at some level in the subsort ordering, might not go away for terms at the kind level. For example, even though addition may be a defined function, we may encounter an arithmetic error expression in a kind of numbers such as

```
(s s zero) + p zero
```

because the predecessor function `p` has been declared on nonzero natural numbers.

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

The last point is that constructors may obey certain equations; that is, they do not have to be *free* constructors. The equations that they may obey (even as constructors, not just in other overloaded variants such as the integer successor function above) may be either equational attributes (such as the `assoc` attribute in the above concatenation operator for strings of natural numbers), or ordinary equations, or both. For example, we can add a sort `NatSet` of finite sets of natural numbers to our `NUMBERS` module by declaring a set union operation `_;_` using equational attributes to declare that it is associative and commutative with identity the empty set, and using an ordinary equation to express idempotency.[6]

```
sort NatSet .
subsort Nat < NatSet .
op empty : -> NatSet [ctor] .
op _;_ : NatSet NatSet -> NatSet [ctor assoc comm id: empty] .
eq N ; N = N .
```

Given an equational specification in which several operators have been declared as constructors by means of the `ctor` attribute and such that the equations are terminating, the *sufficient completeness problem* consists in verifying that the canonical forms of all well-typed ground terms are constructor terms. Intuitively, this means that all defined operations (i.e., those that are not declared as constructors) have been fully defined. Maude's Sufficient Completeness Checker (SCC) can be used to ensure that constructor declarations are really correct, so that all functions are fully defined relative to those constructors. We can take the `NUMBERS` module, incrementally introduced in Chapter 3 and the previous sections of this chapter, to illustrate how the SCC can be used to help the specifier in this regard.

```
fmod NUMBERS is
  sort Zero .
  sorts Nat NzNat .
  subsort Zero NzNat < Nat .
  op zero : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  op sd : Nat Nat -> Nat .
```

[6] Remember that the `idem` attribute cannot be specified together with an `assoc` attribute; therefore idempotency must in this case be specified explicitly by an equation.

```
  ops _+_ _*_ : Nat Nat -> Nat .
  op _+_ : NzNat Nat -> NzNat .
  op p : NzNat -> Nat .

  vars I N M : Nat .
  eq N + zero = N .
  eq N + s M = s (N + M) .
  eq sd(N, N) = zero .
  eq sd(N, zero) = N .
  eq sd(zero, N) = N .
  eq sd(s N, s M) = sd(N, M) .

  sort Nat3 .
  ops 0 1 2 : -> Nat3 .
  op _+_ : Nat3 Nat3 -> Nat3 [comm] .
  vars N3 : Nat3 .
  eq N3 + 0 = N3 .
  eq 1 + 1 = 2 .
  eq 1 + 2 = 0 .
  eq 2 + 2 = 1 .

  sort NatSeq .
  subsort Nat < NatSeq .
  op nil : -> NatSeq [ctor].
  op __ : NatSeq NatSeq -> NatSeq [ctor assoc id: nil] .

  sort NatSet .
  subsort Nat < NatSet .
  op empty : -> NatSet [ctor].
  op _;_ : NatSet NatSet -> NatSet [ctor assoc comm id: empty] .
  eq N ; N = N .
endfm
```

For expository reasons, since the `ctor` declaration had not yet been explained, some operators and constants were declared without the `ctor` attribute when they were introduced in Section 3.6. The SCC reports the first term it finds not reducible to a constructor. In this case, the first such report we get is the following:

```
Maude> (scc NUMBERS .)
Checking sufficient completeness of NUMBERS ...
Warning: This module has equations that are not left-linear which
  will be ignored when checking.
Failure: The term 0 was found to be a counterexample. Since the
  analysis is incomplete, it may not be a real counterexample.
```

We fix this error by adding the `ctor` attribute to the declaration of the constants `0`, `1`, and `2` of sort `Nat3`:

```
ops 0 1 2 : -> Nat3 [ctor].
```

After this declaration is corrected, a more serious bug is found by the SCC, namely,

```
Maude> (scc NUMBERS .)
Checking sufficient completeness of NUMBERS ...
Warning: This module has equations that are not left-linear which
  will be ignored when checking.
```

```
Failure: The term zero * zero was found to be a counterexample.
  Since the analysis is incomplete, it may not be a real
  counterexample.
```

This message shows that the definition of multiplication is incomplete, because we have declared the operator without the `ctor` attribute but we have forgotten the equations defining such operation on natural numbers. For example, we can add the following equations to make up for this omission:

```
eq N * zero = zero .
eq N * s M = (N * M) + N .
```

A further iteration of the SCC on the amended specification shows that the equations for the predecessor operation `p` are missing as well. Since `p` is only defined on nonzero natural numbers, only one equation needs to be added:

```
eq p(s N) = N .
```

The corrected `NUMBERS` module after this analysis (together with some additional declarations introduced in the following sections) is presented in Section 4.9. Here is the tool output on the corrected module:

```
Maude> (scc NUMBERS .)
Checking sufficient completeness of NUMBERS ...
Warning: This module has equations that are not left-linear which
  will be ignored when checking.
Success: NUMBERS is sufficiently complete under the assumption
  that it is weakly-normalizing, confluent, and sort-decreasing.
```

4.4.4 Polymorphic operators

A number of Maude's built-in operators are *polymorphic* in one or more arguments, in the sense that the operator has meaning when these arguments are of any known sort. Examples include Boolean operators such as the conditional, `if_then_else_fi`, which is polymorphic in its second and third arguments, and the equality test `_==_` which is polymorphic in both arguments (see Section 7.1). The user can also define polymorphic operators using the `polymorphic` attribute (abbreviated `poly`). This attribute takes a set of natural numbers enclosed in parentheses that indicates which arguments are polymorphic, with 0 indicating the range. For polymorphic operators that are not constants, at least one argument should be polymorphic to avoid ambiguities. Since there are no polymorphic equations, polymorphic operators are limited to constructors and built-ins. Polymorphic operators are always instantiated with the polymorphic arguments going to the kind level, which further limits their generality. The sort name in a polymorphic position of an operator declaration is purely a place holder—any legal type name could be used. The recommended convention is to use `Universal`.

One reasonable use for polymorphic operators beyond the existing built-ins is to define heterogeneous lists, as follows, where `CONVERSION` denotes a predefined module described in Section 7.9 having types for different numbers as well as strings; this module is imported by means of a `protecting` declaration, which will be explained in Section 6.1.1.

```
fmod HET-LIST is
  protecting CONVERSION .

  sort List .
  op nil : -> List .
```

```
  op __ : Universal List -> List [ctor poly (1)] .
endfm
```

As an example, we can form the following heterogeneous lists:

```
Maude> red 4 "foo" 4.5 1/2 nil .
result List: 4 "foo" 4.5 1/2 nil

Maude> red (4 "foo" nil) 4.5 1/2 nil .
result List: (4 "foo" nil) 4.5 1/2 nil
```

4.4.5 Format

The `format` attribute is intended to control the white space between tokens as well as color and style when printing terms for programming-language-like specifications. Consider the following mixfix syntax operator:

```
op (op_:_->_[_].) : Qid TypeList Type AttrSet -> OpDecl .
```

There are eleven places where white space can be inserted:

```
  op  _  :  _  ->  _  [  _  ]  .
^  ^  ^  ^  ^  ^  ^  ^  ^  ^  ^  ^
```

A format attribute must have an instruction word for each of these places. For example, the formatting specification for the above operator could be chosen to be:

```
[format (d d d d d d s d d s d)]
```

Instruction words are formed from the following alphabet:

`d`	default spacing (cannot be part of a larger word: must occur on its own)
`+`	increment global indent counter
`-`	decrement global indent counter
`s`	space
`t`	tab
`i`	number of spaces determined by indent counter
`n`	newline

Note that, in general, each place may have an entire *word* combining several of the above symbols. We can illustrate how this feature is used in several operators in (submodules of) the `META-LEVEL` module in the file `prelude.maude` (see Chapter 17).

- Each assignment will be printed in a new line, indented one tab.

```
op _<-_ : Variable Term -> Assignment
   [ctor prec 63 format (nt d d d)] .
```

- Each importation after the first one will be printed in a new line, with the current indentation.

```
op __ : ImportList ImportList -> ImportList
   [ctor assoc id: nil format (d ni d)] .
```

- Each kind of declaration in a module will start in a new line, with the current indentation, which is increased by two at the beginning and decreased by two at the end of the module.

```
op fmod_is_sorts_.____endfm : Qid ImportList SortSet
     SubsortDeclSet OpDeclSet MembAxSet EquationSet -> FModule
   [ctor gather (& & & & & & &)
    format (d d d n++i ni d d ni ni ni ni n--i d)] .
```

Whether the format attribute is actually used or not when printing is controlled by the command:

```
set print format on/off .
```

The following additional alphabet can be used to change the text color and style. These colors, perhaps combined with spacing directives, can greatly ease readability, particularly in complex terms for which they can serve as markers. They rely on ANSI escape sequences which are supported by most terminal emulators, most notably the Linux console, Xterm, and Mac Terminal windows, but *not* Emacs shell buffers, unless you use `ansi-color.el`.[7]

`r`	red
`g`	green
`y`	yellow
`b`	blue
`m`	magenta
`c`	cyan
`u`	underline
`!`	bold
`o`	revert to original color and style

By default ANSI escape sequences are suppressed if the environment variable `TERM` is set equal to `dumb` (Emacs does this) or standard output is not a terminal; they are allowed otherwise. This behavior can be overridden by the command line options `-ansi-color` and `-no-ansi-color`.

You are allowed to give a format attribute even if there is no mixfix syntax. In this case the format attribute must have two instruction words, indicating the desired format before and after the operator's name. For example,

```
fmod COLOR-TEST is
  sorts Color ColorList .
  subsort Color < ColorList .
  op red : -> Color [format (r! o)] .
  op green : -> Color [format (g! o)] .
  op blue : -> Color [format (b! o)] .
  op yellow : -> Color [format (yu o)] .
  op cyan : -> Color [format (cu o)] .
  op magenta : -> Color [format (mu o)] .
  op __ : ColorList ColorList -> ColorList [assoc] .
endfm
```

To see the colors in this module, load the `COLOR-TEST` module into Maude and execute the command:[8]

```
Maude> reduce red green blue yellow cyan magenta .
reduce in COLOR-TEST : red green blue yellow cyan magenta .
```

[7] There is a copy of this Emacs Lisp file with the Maude distribution just in case your Emacs distribution lacks it.

[8] Try it in your terminal. The colors are not shown here for obvious reasons.

```
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result ColorList: red green blue yellow cyan magenta
```

Let us consider the following module `FORMAT-DEMO`, where a small programming language is defined.

```
fmod FORMAT-DEMO is
  sorts Variable Expression BoolExp Statement .
  subsort Variable < Expression .
  ops a b c d : -> Variable .
  op 1 : -> Expression .
  op _+_ : Expression Expression -> Expression [assoc comm] .
  op _;_ : Statement Statement -> Statement [assoc prec 50] .
  op _<=_ : Expression Expression -> BoolExp .

  op while_do_od : BoolExp Statement -> Statement
    [format (nir! o r! o++ --nir! o)] .

  op let_:=_ : Variable Expression -> Statement
    [format (nir! o d d d)] .
endfm
```

Note the use of the `format` attribute for operators `while_do_od` and `let_:=_`. Since both represent statements, which should start in a new line, but at the current indentation level, both include `ni` in the instruction words for their first positions; this position also has characters `r!` in both cases, so that they start in boldface red font. Since there is a `o` for the next position, reverting to original color and style, only the first word (`while` and `let`) is shown in red. In the case of `while_do_od`, the condition of the loop starts at the second position. The `do` word is shown in boldface red, and then the indentation counter is incremented, so that the body of the `while_do_od` statement is indented. For the position marking the beginning of `od`, the counter is decremented, so that it appears at the level of `while` in a new line (`n`), in boldface red font (`r!`). The last position reverts the original color and style, although notice that the indentation counter remains the same, so that successive statements will be given the same level of indentation. In the case of `let_:=_`, the three last positions contain only `d` (default spacing), since it is to be presented as a single-line statement in which `let` is shown in boldface red.

We can illustrate the difference between using the format attribute and not using it with the following commands (as before, you should execute the example in your terminal to see the colors).

```
Maude> set print format off .
Maude> parse
        while a <= d do
          let a := a + b ;
          while b <= d do
            let b := b + c ;
            let c := c + 1
          od
        od
        .
Statement: while a <= d do let a := a + b ; while b <= d do let b :=
    b + c ; let c := c + 1 od od

Maude> set print format on .
```

```
Maude> parse
        while a <= d do
          let a := a + b ;
          while b <= d do
            let b := b + c ;
            let c := c + 1
          od
        od
        .
Statement:
while a <= d do
  let a := a + b ;
  while b <= d do
    let b := b + c ;
    let c := c + 1
  od
od
```

For more examples of format attributes, you can see the operator declarations in the module `LTL` (in the file `model-checker.maude`) discussed in Chapter 12, or in the modules `META-TERM` and `META-MODULE` (in the file `prelude.maude`), described in Chapter 17.

4.4.6 Ditto

An operator can have several subsort-overloaded instances. Maude requires that all these instances should have the *same* attributes, *except* for the case of the `ctor` attribute, that may be present in some instances but absent in others (see Section 4.4.3), and/or the `metadata` attribute (see Section 4.5.2). It is for example forbidden to have a subsort-overloaded instance in which an operator is declared `assoc` only, and another such instance in which it is declared `assoc` and `comm`.

The `ditto` attribute can be given to an operator for which another subsort-overloaded instance *has already appeared*, either in the same module or in a submodule. The `ditto` attribute is just a shorthand stating that this operator, being subsort overloaded, should have the same attributes as those appearing explicitly in a previous subsort-overloaded version, except for the `ctor` and `metadata` attributes, which are outside the scope of `ditto`. In this way we can avoid writing out a possibly long attribute list again and again.

It is not allowed to combine `ditto` with other attributes, except for `ctor` and `metadata`. That is, an operator given the `ditto` attribute either has no other explicitly given attributes, or can only have in addition either the `ctor` attribute if it is a constructor, or a `metadata` attribute, or both the `ctor` and `metadata` attributes. Furthermore, it is forbidden to use `ditto` on the first declared instance of an operator, since this is nonsensical.

In our numbers module we can add equational attributes to the declarations of `_+_` and `_*_`, and then use `ditto` to declare the same attributes in other subsort-overloaded versions.

```
ops _+_ _*_ : Nat Nat -> Nat [assoc comm].
op _+_ : NzNat Nat -> NzNat [ditto] .
op _*_ : NzNat NzNat -> NzNat [ditto] .
```

For an example making extensive use of the `ditto` attribute see the `LTL-SIMPLIFIER` module (in the file `model-checker.maude`), discussed in Chapter 12.

4.4.7 Operator evaluation strategies

If a collection of equations is Church-Rosser and terminating, given an expression, no matter how the equations are used from left to right as simplification rules, we will always reach the same final result. However, even though the final result may be the same, some orders of evaluation can be considerably more efficient than others. More generally, we may be able to achieve the termination property provided we follow a certain order of evaluation, but may lose termination when any evaluation order is allowed. It may therefore be useful to have some way of controlling the way in which equations are applied by means of strategies.

In general, given an expression $f(t_1, \ldots, t_n)$ we can try to evaluate it to its reduced form in different ways, such as:

- first obtaining the reduced form of all the t_i and then applying equations for f at the top of the term; this is called a *bottom-up*, or *eager* strategy;
- evaluating only some of the arguments, and then trying to evaluate at the top with equations for f; for example, an `if_then_else_fi` operator will typically be evaluated by evaluating first the first argument, and then the `if_then_else_fi` operator at the top;
- trying to evaluate the top of the term first, and then, if this fails, either not evaluating the subterms at all, or trying to evaluate only some of them, that is, some kind of *lazy* evaluation strategy.

Typically, a functional language is either eager, or lazy with some strictness analysis added for efficiency, and the user has to live with whatever the language provides. Maude adopts OBJ3's [79] flexible method of user-specified *evaluation strategies* on an operator-by-operator basis, adding some improvements to the OBJ3 approach to ensure a correct implementation [58].

For an n-ary operator f an evaluation strategy is specified as a list of numbers from 0 to n ending with 0. The nonzero numbers denote argument positions, and a 0 indicates evaluation at the top of the given function symbol. The strategy then specifies what argument positions must be simplified (in the order indicated by the list) before attempting simplification at the top with the equations for the top function symbol. In functional programming terminology, the argument positions to be evaluated are usually called *strict* argument positions, so we can view an evaluation strategy as a flexible, user-definable way of specifying strictness requirements on argument positions. In the simplest case, a strategy consists of a list of nonzero numbers followed by a 0, so that some arguments are treated strictly and then the function symbol's equations are applied. For example, in Maude, if no strategy is specified, all argument positions are assumed strict, so that for f with n argument positions its default strategy is $(1\,2\,...\,n\,0)$; this is the "eager evaluation" case. The opposite extreme is a form of lazy evaluation such as the lazy append operator in the `SIEVE` example below. This operator has strategy (0), thus only equations at the top are tried during evaluation.

The syntax to declare an n-ary operator with strategy $(i_1 \ \ldots \ i_k \ 0)$, where $i_j \in \{0, \ldots, n\}$ for $j = 1, \ldots, k$, is

`op` ⟨*OpName*⟩ `:` ⟨*Sort-1*⟩ `...` ⟨*Sort-n*⟩ `->` ⟨*Sort*⟩ `[strat (`i_1 `...` i_k `0)] .`

As a simple example consider the operators `_and-then_` and `_or-else_` in the module `EXT-BOOL`, that can be found in the file `prelude.maude` (see Section 7.1).

```
fmod EXT-BOOL is
  protecting BOOL .
  op _and-then_ : Bool Bool -> Bool
```

```
    [strat (1 0) gather (e E) prec 55] .
  op _or-else_ : Bool Bool -> Bool
    [strat (1 0) gather (e E) prec 59] .
  var B : [Bool] .
  eq true and-then B = B .
  eq false and-then B = false .
  eq true or-else B = true .
  eq false or-else B = B .
endfm
```

These operators are computationally more efficient versions of Boolean conjunction and disjunction that avoid evaluating the second of the two Boolean subterms in their arguments when the result of evaluating the first subterm provides enough information to compute the conjunction or the disjunction. For example, letting `B:[Bool]` stand for an arbitrary Boolean expression

```
Maude> red false and-then B:[Bool] .
result Bool: false
```

while if `B:[Bool]` does not evaluate to `true` or `false`, then `false and B:[Bool]` does not evaluate to `false`, and if evaluation of `B:[Bool]` does not terminate then neither will evaluation of `false and B:[Bool]`.

If some of the argument positions are never mentioned in some of the operator strategies, the notion of canonical form becomes now *relative* to the given strategies and may not coincide with the standard notion. Let us consider as a simple example the following two functional modules, which we have displayed side-by-side to emphasize their only difference, namely, the evaluation strategy associated to the operator `g`.

```
fmod STRAT-EX1 is
  sort S .
  ops a b : -> S .
  op g : S -> S .
  eq a = b .
endfm
```

```
fmod STRAT-EX2 is
  sort S .
  ops a b : -> S .
  op g : S -> S [strat(0)] .
  eq a = b .
endfm
```

The canonical form of the term `g(a)` in `STRAT-EX1` is `g(b)`, but in `STRAT-EX2` it is `g(a)` itself, because the equation cannot be applied inside the term due to the lazy strategy `strat(0)` of the operator `g`.

This may be just what we want, since we may be able to achieve termination to a canonical form relative to some strategies in cases when the equations may be nonterminating in the standard sense. More generally, operator strategies may allow us to compute with infinite data structures which are evaluated on demand, such as the following formulation of the *sieve of Eratosthenes*, which finds all prime numbers using lazy lists.

The infinite list of primes is obtained from the infinite list of all natural numbers greater than 1 by filtering out all the multiples of numbers previously taken. Thus, first we take 2 and delete all even numbers greater than 2; then we take 3 and delete all the multiples of 3 greater than 3; and so on. The operation `nats-from_` generates the infinite list of natural numbers starting in the given argument; the operation `filter_with_` is used to delete all the multiples of the number given as second argument in the list provided as first argument; and the operation `sieve_` is used to iterate this process with successive numbers.

Of course, since we are working with infinite lists, we cannot obtain as a result an infinite list. Such an infinite structure is only shown partially by means of the operation `show_upto_`, which shows only a finite prefix of the whole infinite list. Moreover, the generation and filtering processes have to be done in a lazy way. This is accomplished by giving to the list constructor

`_._` a lazy strategy `strat(0)` that avoids evaluating inside the term, and using an operation `force` with an eager strategy `strat(1 2 0)` to "force" the evaluation of elements inside the list. Specifically, in order to apply the first equation, we must evaluate the arguments `L` and `S` before reconstructing the list `L . S` in the righthand side.

`NAT` denotes the predefined module of natural numbers and arithmetic operations on them (see Section 7.2), which is imported by means of a `protecting` declaration, explained in Section 6.1.1. Note the use of the symmetric difference operator `sd` (see Section 7.2) to decrement `I` in the third equation, and the successor operator `s_` to increment `I` in the sixth equation.

```
fmod SIEVE is
  protecting NAT .
  sort NatList .
  subsort Nat < NatList .
  op nil : -> NatList .
  op _._ : NatList NatList -> NatList [assoc id: nil strat (0)] .
  op force : NatList NatList -> NatList [strat (1 2 0)] .
  op show_upto_ : NatList Nat -> NatList .
  op filter_with_ : NatList Nat -> NatList .
  op nats-from_ : Nat -> NatList .
  op sieve_ : NatList -> NatList .
  op primes : -> NatList .

  vars P I E : Nat .
  vars S L : NatList .

  eq force(L, S) = L . S .
  eq show nil upto I = nil .
  eq show E . S upto I
    = if I == 0
      then nil
      else force(E, show S upto sd(I, 1))
      fi .
  eq filter nil with P = nil .
  eq filter I . S with P
    = if (I rem P) == 0
      then filter S with P
      else I . filter S with P
      fi .
  eq nats-from I = I . nats-from (s I) .
  eq sieve nil = nil .
  eq sieve (I . S) = I . sieve (filter S with I) .
  eq primes = sieve nats-from 2 .
endfm
```

We can then evaluate expressions in this module with the `reduce` command (see Sections 4.9 and 23.2). For example, to compute the list of the first ten prime numbers we evaluate the expression:

```
Maude> reduce show primes upto 10 .
result NatList: 2 . 3 . 5 . 7 . 11 . 13 . 17 . 19 . 23 . 29
```

In the case of associative or commutative binary operators, evaluation strategies might reduce some arguments that the user does not expect to be reduced. The reason is that in such cases terms represent equivalence classes and it might be quite hard to say what is the first or the second argument. The adopted solution is that mentioning either argument implies both.

The paper [58] documents the operational semantics and the implementation techniques for Maude's operator evaluation strategies in much more detail. The mathematical semantics of a Maude functional module having operator evaluation strategies is documented in [81] and is further discussed in Section 4.7.

Of course, operator evaluation strategies, while quite useful, are by design restricted in their scope of applicability to *functional modules*.[9] As we shall see in Chapter 5, *system modules*, specifying rewrite theories that are not functional, need not be Church-Rosser or terminating, and require much more general notions of strategy. Such general strategies are provided by Maude using reflection by means of *internal strategy languages*, in which strategies are defined by rewrite rules at the metalevel (see Section 17.7). However, as discussed in Section 4.4.9, specifying *frozen* arguments in operators restricts the rewrites allowed with rules in a system module (as opposed to equations) in a way quite similar to how operator evaluation strategies restrict the application of equations in a functional module.

4.4.8 Memo

If an operator is given the `memo` attribute, this instructs Maude to *memoize* the results of equational simplification (that is, the canonical forms) for those subterms having that operator at the top. This means that when the canonical form of a subterm having that operator at the top is obtained, an entry associating to that subterm its canonical form is stored in the memoization table for this operator. Whenever the Maude interpreter encounters a subterm whose top operator has the `memo` attribute, it looks to see if its canonical form is already stored. If so, that result is used; otherwise, equational simplification proceeds according to the operator's strategy. Giving to some operators the `memo` attribute allows trading off space for time in equational simplifications: more space is needed, but if subcomputations involving memoized operators have to be repeated many times, then a computation may be substantially sped up, provided that the machine's main memory limits are not exceeded.

An operator's `memo` attribute and its user's specified or default evaluation strategy (see Section 4.4.7) may interact with each other, impacting the size of the memoization table. The issue is how many entries for different subterms, all having the same canonical form, may be possibly stored in the memoization table. If the operator has the default, bottom-up strategy, the answer is: *only one such entry is possible*. For other strategies, different terms having the same canonical form may be stored, making the memoization table bigger. For example, using the default strategy `(1 2 0)` for a memoized operator `f`, only subterms of the form `f(v, v')` with `v` and `v'` fully reduced to canonical form (up to the strategies given for all operators) will be mapped to their corresponding canonical forms. This is because, with the default strategy, equational simplification at the top of `f` can only happen after all its arguments are in canonical form. For other operator strategies this uniqueness may be lost, even when evaluating just one subterm involving `f`. For example, if `f`'s strategy is `(0 1 2 0)`, then both the starting term `f(t, t')` and the term `f(v, v')` (where `v` and `v'` are, respectively, the canonical forms of `t` and `t'`) will be mapped to the final result, since the strategy specifies rewriting at the top twice. That is, each time the operator's strategy calls for rewriting at the top, Maude will add the current version of the term to the set of terms that will be mapped to the final result. Furthermore, other terms of the form `f(u, u')`, with `u` and `u'` having also `v` and `v'` as their canonical forms may appear in other subcomputations, and will then also be stored in the memoization table.

In general, whenever an application will perform an operation many times, it may be useful

[9] More precisely, the scope of applicability of operator evaluation strategies is restricted to functional modules and to the *equational* part of system modules.

to give that operator the `memo` attribute. This may be due to the high frequency with which the operator is called by other operators in a given application, or to the highly recursive nature of the equations defining that operator. For example, the recursive definition of the Fibonacci function is given as follows, where `NAT` denotes the predefined module of natural numbers and arithmetic operations on them (as described in Section 7.2), which is imported by means of a `protecting` declaration (see Section 6.1.1).

```
fmod FIBONACCI is
  protecting NAT .
  op fibo : Nat -> Nat .

  var N : Nat .
  eq fibo(0) = 0 .
  eq fibo(1) = 1 .
  eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

Due to the highly recursive nature of this definition of `fibo`, the evaluation of an expression like `fibo(50)` will compute many calls to the same instances of the function again and again, and will expand the original term into a whole binary tree of additions before collapsing it to a number. The exponential number of repeated function calls makes the evaluation of `fibo` with the above equations very inefficient, requiring over 61 billion rewrite steps for `fibo(50)`:

```
Maude> red fibo(50) .
reduce in FIBONACCI : fibo(50) .
rewrites: 61095033220 in 132081000ms cpu (145961720ms real)
  (462557 rews/sec)
result NzNat: 12586269025
```

If we instead give the Fibonacci function the `memo` attribute,

```
  op fibo : Nat -> Nat [memo] .
```

the change in performance is quite dramatic:

```
Maude> red fibo(50) .
reduce in FIBONACCI : fibo(50) .
rewrites: 148 in 0ms cpu (0ms real) (~ rews/sec)
result NzNat: 12586269025

Maude> red fibo(100) .
reduce in FIBONACCI : fibo(100) .
rewrites: 151 in 0ms cpu (1ms real) (~ rews/sec)
result NzNat: 354224848179261915075

Maude> red fibo(1000) .
reduce in FIBONACCI : fibo(1000) .
rewrites: 2701 in 0ms cpu (11ms real) (~ rews/sec)
result NzNat: 43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

In some cases we may introduce a *constant operator* as an abbreviation for a possibly complex expression that may require a substantial number of equational simplification steps to be reduced to canonical form; furthermore, the operator may be used repeatedly in different

subcomputations. In such cases one can declare a constant operator, give it the `memo` attribute, and give an equation defining it to be equal to the expression of interest. For example, suppose we have defined a search space with initial state `myState` and a function `findAnswer` to search the space for a state satisfying some property. Then we can name the search result and use it again without redoing an expensive computation as follows:

```
op myAns : -> Answer [memo] .
eq myAns = findAnswer(myState) .
```

Maude will then remember the result of rewriting the constant in the memoization table for that operator and will not repeat the work until the memoization tables are cleared. Memoization tables for the current module can be cleared explicitly by the command

```
do clear memo .
```

Automatic clearing before each top level rewriting command can be turned on and off with

```
set clear memo on .
set clear memo off .
```

By default, `set clear memo` is off.

4.4.9 Frozen arguments

The `frozen` attribute is only meaningful for system modules (see Chapter 5) that may have both rules and equations. It has no direct effect for functional modules having only equations and memberships: it can only have an *indirect* effect if the functional module is later imported by a system module. For this reason, examples of the use of frozen operators are postponed to Chapter 5.

Given a system module `M`, by declaring a given operator, say `f`, as `frozen`, rewriting with rules is always forbidden in all proper subterms of a term having `f` as its top operator. However, it may still be possible to rewrite that term at the top, provided rules having `f` as the top symbol of their lefthand side exist in `M`. To specify that all the arguments of an operator are frozen, one includes the attribute `frozen` in the operator's list of attributes; for example,

```
op f : S1 ... Sn -> S [frozen] .
```

The freezing idea can be generalized, so that only specific *argument positions* of the operator `f` are frozen. For example, in a system module specifying the semantics of a programming language with rewrite rules, we may want to specify a sequential composition operator `_;_` as frozen in its second argument, but not in the first argument, so as to prevent any execution of the second program fragment of the composition from happening before the first fragment has been fully evaluated. We can specify this by stating

```
op _;_ : Program Program -> Program [frozen (2)] .
```

More generally, if the list of argument positions in an operator `f` is $1 \ldots n$, then we can freeze any sublist of argument positions, say $i_1 \ldots i_m$, by declaring,

```
op f : S1 ... Sn -> S [frozen (i1 ... im)] .
```

Of course, if the actual list of specified positions is $1 \ldots n$ itself, then this is equivalent to the first mode of declaring the `frozen` attribute for `f` without listing any positions.

As for operator evaluation strategies (see Section 4.4.7), in the case of associative or commutative binary operators mentioning either argument in the list of frozen positions implies both.

4.4.10 Special

Many operators in predefined modules (see Chapters 7 and 17) have the `special` attribute in their declarations. This means that they are to be treated as *built-in* operators, so that, instead of having the standard treatment of any user-defined operator, they are associated with appropriate C++ code by "hooks" which are specified following the `special` attribute identifier.

For example, the file `prelude.maude` contains a predefined module `NAT` for natural numbers and usual operations on them (see Section 7.2). Among others, the declarations in the `NAT` module for the operations of addition and of quotient of integer division, and for a less than predicate are the following:

```
op _+_ : NzNat Nat -> NzNat
     [assoc comm prec 33
      special (id-hook ACU_NumberOpSymbol (+)
               op-hook succSymbol (s_ : Nat ~> NzNat))] .
op _+_ : Nat Nat -> Nat [ditto] .

op _quo_ : Nat NzNat -> Nat
     [prec 31 gather (E e)
      special (id-hook NumberOpSymbol (quo)
               op-hook succSymbol (s_ : Nat ~> NzNat))] .

op _<_ : Nat Nat -> Bool
     [prec 37
      special (id-hook NumberOpSymbol (<)
               op-hook succSymbol (s_ : Nat ~> NzNat)
               term-hook trueTerm (true)
               term-hook falseTerm (false))] .
```

Notice that the `special` attribute exists in order to bind Maude syntax to built-in C++ functionality. It is absolutely not for users to mess with and it is absolutely not backwards compatible; this is why Maude will sometimes crash or become unstable if the prelude from a different version is loaded. For the same reason, other operator attributes that appear together with `special` in an operator declaration cannot be modified either.

4.5 Statement attributes

In a functional module, statements are equations and membership axioms, conditional or not. Any such statement may have associated *attributes*. Currently five attributes are available: `label`, `metadata`, `nonexec`, `owise`, and `print`. The attributes `label`, `metadata`, `nonexec`, and `print` can also be used on rules in system modules. Moreover, the attribute `metadata` can also be associated to operator declarations.

4.5.1 Labels

The `label` attribute must be followed by an identifier. Statement labels can be used for tracing and debugging and at the metalevel to name particular axioms. In our numbers example we could label the axiom for idempotency for natural number sets

```
eq N ; N = N [label natset-idem] .
```

Syntactic sugar for labels generalizing the Maude 1 style for rule labels is also supported. Then the above label could have also been written

```
eq [natset-idem] : N ; N = N .
```

4.5.2 Metadata

The `metadata` attribute must be followed by a string (that is, by a data element in the `STRING` module, see Section 7.8). The metadata attribute is intended to hold data about the statement in whatever syntax the user cares to create/parse. It is like a comment that is carried around with the statement. Usual string escape conventions apply. For example, we could add the distributive law

```
eq (N + M) * I = (N * I) + (M * I) [metadata "distributive law"] .
```

with the comment documenting that this is the distributive law.

The `metadata` attribute can also be associated to operator declarations. Note that, like `ctor`, `metadata` is attached to a specific operator declaration and not to the (possibly overloaded) operator itself. Thus:

- two subsorted overloaded declarations may have different `metadata` attributes,
- a `metadata` attribute is not copied by the `ditto` attribute (see Section 4.4.6), and
- a declaration may have a `metadata` attribute as well as a `ditto` attribute.

Under these conditions, the following ad-hoc example is therefore legal:

```
fmod METADATA-EX is
  sorts Foo Bar .
  subsort Foo < Bar .
  op f : Foo -> Foo [memo metadata "f on Foos"] .
  op f : Bar -> Bar [ditto metadata "f on Bars"] .
endfm
```

4.5.3 Nonexec

The `nonexec` attribute allows the user to include statements in a module that are ignored by the Maude rewrite engine. For example we could make the distributive law nonexecutable as follows.

```
eq (N + M) * I = (N * I) + (M * I)
  [nonexec metadata "distributive law"] .
```

Similarly, a rule can be declared with the `nonexec` attribute in a system module.

Although nonexecutable from the point of view of Core Maude, such statements are part of the semantics of the module and can for example be used at the metalevel for controlled execution or theorem proving purposes.

4.5.4 Otherwise

Sometimes, in the definition of an operation by equations, there are certain cases that can be easily defined by equations, and then some remaining case or cases that it is more difficult or cumbersome to define. One would in such situations like to say, *otherwise*, that is, in all remaining cases not covered by the above equations, do so and so.[10]

Consider, for example, the problem of membership of a natural number in a finite set of numbers.

[10] Indeed, several languages have conventions of this kind, including ASF+SDF [42].

```
op _in_ : Nat NatSet -> Bool .
```

The easy part is to define when a number belongs to a set:

```
var N : Nat .
var NS : NatSet .
eq N in N ; NS = true .
```

It is somewhat more involved to define when it *does not* belong. A simple way is to use the `otherwise` (abbreviated `owise`) attribute and give the additional equation:

```
eq N in NS = false [owise] .
```

The intuitive operational meaning is clear: if the first equation does not match, then the number in fact is not in the set, and the predicate should be false. But what is the *mathematical* meaning? That is, how can we interpret the meaning of the second equation so that it becomes a useful *shorthand* for an ordinary equation? After all, the second equation, as given, is even more general than the first and in direct contradiction with it. We of course should reject any constructs that violate the logical semantics of the language.

Fortunately, there is nothing to worry about, since the `owise` attribute is indeed a shorthand for a corresponding *conditional* equation. We first explain the idea in the context of this example and then discuss the general construction. The idea is that, whether an equation, or a set of equations, defining the meaning of an operation f match a given term, is itself a property defined by a predicate, say $enabled_f$, which is effectively definable by equations. In our example we can introduce a predicate `enabled-in`, telling us when the first equation applies, by just giving its lefthand side arguments as the predicate's arguments:

```
op enabled-in : [Nat] [NatSet] -> [Bool] .
eq enabled-in(N, N ; NS) = true .
```

Note that we do not have to define when the `enabled-in` predicate is *false*. That is, this predicate is really defined on the kind `[Bool]`. Our second `owise` equation is simply a convenient shorthand for the *conditional* equation

```
ceq N in NS = false if enabled-in(N, NS) =/= true .
```

This is just a special case of a completely general *theory transformation* that translates a specification containing equations with the `owise` attribute into a semantically equivalent specification with no such attributes at all. A somewhat subtle aspect of this transformation[11] is the interaction between `owise` equations and the operator evaluation strategies discussed in Section 4.4.7. Suppose that an `owise` equation was used in defining the semantics of an operator f. If f was (implicitly or explicitly) declared with a strategy, say,

$$f : s_1 \dots s_n \to s \quad [\texttt{strat} \ (i_1 \dots i_k 0)] \ .$$

then, the $enabled_f$ predicate should be defined with the *same* strategy,

$$enabled_f : [s_1] \dots [s_n] \to \texttt{[Bool]} \quad [\texttt{strat} \ (i_1 \dots i_k 0)] \ .$$

This will make sure that the reduction of f's arguments prior to applying equations for f—including the equations that will be introduced in our transformation to replace the `owise` equations—takes place in exactly the same way for f and for $enabled_f$, so that failure of matching the normal equations is correctly captured by the failure of the $enabled_f$ predicate. Furthermore, as we shall see, after the failure of matching the non-`owise` equations, the matching substitution obtained when we apply the desugared version of an `owise` equation will then

[11] We thank Joseph Hendrix for pointing out this subtlety.

properly take into account the evaluation of those arguments of f specified by f's evaluation strategy.

In general, if we are defining the equational semantics of an operation $f : s_1 \dots s_n \to s$ and we have given a partial definition of that operation by (possibly conditional) equations

$$\begin{aligned} f(u_1^1, \dots, u_n^1) &= t_1 \ \textit{if}\ C_1 \\ &\dots \\ f(u_1^m, \dots, u_n^m) &= t_m \ \textit{if}\ C_m \end{aligned}$$

then we can give one or more `owise` equations defining the function in the remaining cases by means of equations of the form

$$\begin{aligned} f(v_1^1, \dots, v_n^1) &= t'_1 \ \textit{if}\ C'_1 \ \texttt{[owise]} \\ &\dots \\ f(v_1^k, \dots, v_n^k) &= t'_k \ \textit{if}\ C'_k \ \texttt{[owise]} \end{aligned}$$

We can view such `owise` equations as shorthand notation for corresponding ordinary conditional equations of the form

$$\begin{aligned} f(y_1, \dots, y_n) = t'_1 \quad & \textit{if} \quad \textit{enabled}_f(y_1, \dots, y_n) \neq \texttt{true} \\ & \wedge \quad \textit{enabled}_f(v_1^1, \dots, v_n^1) := \textit{enabled}_f(y_1, \dots, y_n) \\ & \wedge \quad C'_1 \\ & \dots \\ f(y_1, \dots, y_n) = t'_k \quad & \textit{if} \quad \textit{enabled}_f(y_1, \dots, y_n) \neq \texttt{true} \\ & \wedge \quad \textit{enabled}_f(v_1^k, \dots, v_n^k) := \textit{enabled}_f(y_1, \dots, y_n) \\ & \wedge \quad C'_k \end{aligned}$$

where the variables $y_1, \dots, y_n$ are *fresh new variables* not appearing in any of the above `owise` equations, and with y_i of kind $[s_i]$, $1 \leq i \leq n$. All this assumes that in the transformed specification we have declared the predicate $\textit{enabled}_f : [s_1] \dots [s_n] \to$ `[Bool]`, with the same evaluation strategy as f. Note the somewhat subtle use of the matching equations (see Section 4.3) $\textit{enabled}_f(v_1^j, \dots, v_n^j) := \textit{enabled}_f(y_1, \dots, y_n)$, $1 \leq j \leq k$, in the conditions. Since f and $\textit{enabled}_f$ have the same strategy, after the arguments of the matching instance of the expression $\textit{enabled}_f(y_1, \dots, y_n)$ become evaluated according to the strategy, we are then able to match $\textit{enabled}_f(v_1^j, \dots, v_n^j)$ to that result, obtaining the desired substitution for the variables of the lefthand side of the j^{th} `owise` equation. That is, we obtain the same substitution as the one we would have obtained matching $f(v_1^j, \dots, v_n^j)$ to the same subject term after its subterms under f had been evaluated according to f's strategy.

Of course, the semantics of the $\textit{enabled}_f$ predicate is defined in the expected way by the equations

$$\begin{aligned} \textit{enabled}_f(u_1^1, \dots, u_n^1) &= \texttt{true} \ \textit{if}\ C_1 \ . \\ &\dots \\ \textit{enabled}_f(u_1^m, \dots, u_n^m) &= \texttt{true} \ \textit{if}\ C_m \ . \end{aligned}$$

The possibility of using multiple `owise` equations allows us to simplify definitions of functions defined by cases on data with nested structure. Here is a simple, if silly, example in which the sort `R` has elements `a(n)` and `b(n)`, for natural numbers `n`, and the sort `S` has elements

`g(r)` and `h(r)`, with `r` of sort `R`. The operation `f` treats constructors `g` and `h` differently, distinguishing only whether the subterm of sort `R` is constructed by `a` or not. Again, the predefined module `NAT` of natural numbers (Section 7.2) is imported by means of a `protecting` declaration (Section 6.1.1).

```
fmod OWISE-TEST1 is
  protecting NAT .

  sorts R S .
  op f : S Nat -> Nat .
  ops g h : R -> S .
  ops a b : Nat -> R .

  var  r : R .
  vars m n : Nat .
  eq f(g(a(m)), n) = n .
  eq f(h(a(m)), n) = n + 1 .
  eq f(g(r), n) = 0 [owise] .
  eq f(h(r), n) = 1 [owise] .
endfm
```

The four cases are illustrated by the following reductions.

```
Maude> red f(g(a(0)), 3) .
result NzNat: 3

Maude> red f(g(b(0)), 3) .
result Zero: 0

Maude> red f(h(b(0)), 3) .
result NzNat: 1

Maude> red f(h(a(0)), 3) .
result NzNat: 4
```

The subtle interaction between `owise` equations and operator evaluation strategies can be illustrated by the following example:

```
fmod OWISE-TEST2 is
  sort Foo .
  ops a b c d : -> Foo .
  op f : Foo -> Foo [strat (0 1 0)] .
  op g : Foo -> Foo [strat (0)] .

  var X : Foo .
  eq b = c .
  eq f(a) = d .
  eq f(X) = g(X) [owise] .
endfm
```

Now consider the term `f(b)`. Intuitively, one could expect that, given that the first equation for `f` cannot be applied to this term, the `owise` equation is applied obtaining the term `g(b)`, and this is then expected to be the final result of the reduction, because the strategy `(0)` for `g` forbids evaluating its argument. However, as we can see in the following reduction, this is not the case.

```
Maude> red f(b) .
result Foo: g(c)
```

The result is `g(c)`, because the `owise` equation is not considered until after evaluating the final 0 in the strategy for `f`, and by then `f(b)` is simplified to `f(c)` as instructed by the 1 in such strategy; then, the `owise` equation applied to `f(c)` produces `g(c)`.

It can be interesting to consider the semantically equivalent transformed specification:

```
fmod OWISE-TEST2-TRANSFORMED is
  sort Foo .
  ops a b c d : -> Foo .
  op f : Foo -> Foo [strat (0 1 0)] .
  op enabled-f : Foo -> Bool [strat (0 1 0)] .
  op g : Foo -> Foo [strat (0)] .

  vars X Y : Foo .
  eq b = c .
  eq f(a) = d .
  eq enabled-f(a) = true .
  ceq f(Y) = g(X)
    if enabled-f(Y) =/= true /\ enabled-f(X) := enabled-f(Y) .
endfm

Maude> red f(b) .
result Foo: g(c)
```

where, as pointed out in our comments on the general transformation, the fact that `enabled-f` has the same strategy as `f` and the use of the matching equation

```
enabled-f(X) := enabled-f(Y)
```

are crucial for obtaining a semantically equivalent specification.

4.5.5 Print

The `print` attribute allows the user to specify information to be printed when a statement (equation, membership axiom, or rule) is executed. A print attribute declaration looks like

```
eq f(X) = b [print "X = " X] .
```

The keyword `print` is followed by a possibly empty list of items where each item is either a string constant or a variable. Mentioned variables must actually occur in the statement. If a non-occurring variable appears as a `print` item, it will be pruned and Maude will issue a warning.

Here is an example that uses the `print` attribute to track calls to a recursive function that reverses a list.

```
fmod PRINT-ATTRIBUTE-EX is
  sorts Foo FooList .
  ops a b : -> Foo [ctor] .
  subsort Foo < FooList .
  op nil : -> FooList [ctor] .
  op __ : FooList FooList -> FooList [ctor assoc id: nil] .

  op reverse : FooList -> FooList .
  eq reverse(nil) = nil .
```

```
  eq reverse(foo:Foo fl:FooList) = reverse(fl:FooList) foo:Foo
       [print "first = " foo:Foo ", rest = " fl:FooList] .
endfm
```

Maude will only use the `print` attribute in print attribute mode, which is off by default. Thus to run the above example (after loading it into Maude) it is necessary to execute the following command (see Section 23.9).

```
Maude> set print attribute on .
```

Then reducing the expression `reverse(a b a b)` results in the following output:

```
Maude> red in PRINT-ATTRIBUTE-EX : reverse(a b a b) .
reduce in PRINT-ATTRIBUTE-EX : reverse(a b a b) .
first = a, rest = b a b
first = b, rest = a b
first = a, rest = b
first = b, rest = nil
rewrites: 5 in 0ms cpu (0ms real) (18587 rewrites/second)
result FooList: b a b a
```

The `print` attribute is an alternative to tracing (see Section 20.1.1) to find out which statements Maude is executing. It allows the user control of what information is printed. It is also a nice way to show what is going on in demos.

4.6 Admissible functional modules

The `nonexec` attribute allows us to include arbitrary equations or memberships, conditional or not, in a functional module and likewise in a functional theory (see Section 6.3.1). Any such statement is then disregarded for purposes of execution by the Maude engine: it can only be used in a controlled way at the metalevel. But what about all the other statements? That is, what requirements should be imposed on *executable* equations and memberships so that they can be given an operational interpretation and can be executed by the Maude engine?

The intuitive idea is that we want to use such equations as *simplification rules* from left to right to reach a single final result or canonical form. For this purpose, the executable equations and memberships (that is, all statements not having the `nonexec` attribute) should be Church-Rosser and terminating (*modulo* the equational attributes declared in the module) in the sense explained in Section 4.7 below. This guarantees that, given a term t, all chains of equational simplification using those equations and memberships end in a unique canonical form (again, modulo the equational attributes). Furthermore, under the preregularity assumption (see Section 3.8), such a canonical form has the *smallest sort possible* in the subsort ordering.

The traditional requirement in this context is that, given a conditional equation[12] $t = t'$ *if* $C_1 \wedge \ldots \wedge C_n$, the set of variables appearing in t contains those appearing in both t' and in the conditions C_i. In Maude, this requirement is relaxed to support matching equations in conditions (see Section 4.3) which can introduce new variables not present in t. Specifically, all executable conditional equations in a Maude functional module `M` have to satisfy the following *admissibility requirements*, ensuring that all the extra variables will become instantiated by matching:

[12] For the purposes of this discussion we can regard unconditional equations as the special case of conditional equations with empty condition, or with the condition $true = true$.

1.
$$vars(t') \subseteq vars(t) \cup \bigcup_{j=1}^{n} vars(C_j).$$

2. If C_i is an equation $u_i = u_i'$ or a membership $u_i : s$, then
$$vars(C_i) \subseteq vars(t) \cup \bigcup_{j=1}^{i-1} vars(C_j).$$

3. If C_i is a matching equation $u_i := u_i'$, then u_i is an `M`-pattern and
$$vars(u_i') \subseteq vars(t) \cup \bigcup_{j=1}^{i-1} vars(C_j).$$

In a similar way, all executable conditional memberships $t : s$ *if* $C_1 \wedge \ldots \wedge C_n$ must satisfy conditions 2–3 above.

In summary, therefore, we expect all executable equations and memberships in a functional module (and also in a system module) to be Church-Rosser and terminating (see Section 4.7 below, and [17, Section 10.1]) and to satisfy the above admissibility requirements.

4.7 Matching and equational simplification

Although this section and the next are quite technical, and it may be possible to skip them in a first reading, they introduce the concepts of *matching* and *equational simplification* that are essential to understand how Maude works. Therefore, we advise the reader to come back to them as needed to gain a better understanding of those concepts.

Recall from Section 4.3 that a functional module defines an equational theory $(\Sigma, E \cup A)$ in membership equational logic, with A the equations specified as equational attributes in operators (see Section 4.4.1), and E the (possibly conditional) equations and memberships specified as statements.

Ground terms in the signature Σ form a Σ-algebra denoted T_Σ. Similarly, equivalence classes of terms modulo $E \cup A$ define the Σ-algebra denoted $T_{\Sigma/E \cup A}$, which is the *initial model* for the theory $(\Sigma, E \cup A)$ specified by the module [106].

Given a set X of variables, we can add them to the signature Σ as new constants, and get in this way a term algebra $T_\Sigma(X)$ where now the terms may have variables in X.

Given a set X of variables, each having a given kind, a (ground) *substitution* is a kind-preserving function $\sigma : X \longrightarrow T_\Sigma$. Such substitutions may be used to represent assignments of terms in T_Σ to the variables in X, and also assignments of elements in $T_{\Sigma/E \cup A}$ to such variables by σ picking up a representative of the corresponding $E \cup A$-equivalence class. For example, a very natural choice is to assign to each x in X a term $\sigma(x)$ which is in *canonical form* according to $E \cup A$. Furthermore, under the preregularity, Church-Rosser, and termination assumptions (more on this below) this canonical form will have a *least sort*. Therefore, we may allow each variable x in X to have either a kind or a sort assigned to it, and can call the substitution σ *well-sorted* relative to $E \cup A$ if the least sort of $\sigma(x)$ is smaller or equal to that of x. By substituting terms for variables (as indicated by σ) in the usual way, a substitution $\sigma : X \longrightarrow T_\Sigma$ is extended to a homomorphic function on terms $\sigma : T_\Sigma(X) \longrightarrow T_\Sigma$ that we denote with the same name.

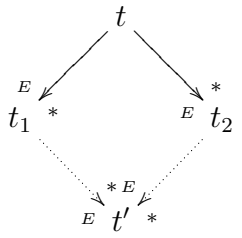


Figure 4.1: Confluence diagram

Given a term $t \in T_\Sigma(X)$, corresponding to the lefthand side of an oriented equation, and a subject ground term $u \in T_\Sigma$, we say that t *matches*[13] u if there is a substitution σ such that $\sigma(t) \equiv u$, that is, $\sigma(t)$ and u are syntactically equal terms.

For an oriented Σ-equation $l = r$ to be used in equational simplification, it is required that all variables in the righthand side r also appear among the variables of the lefthand side l. In the case of a conditional equation $l = r$ *if cond*, this requirement is relaxed, so that more variables can appear in the condition *cond*, provided that they are introduced by matching equations according to the admissibility requirements in Section 4.6; then the variables in the righthand side r must be among those in the lefthand side l or in the condition *cond*. Under this assumption, given a theory (Σ, E) a term t *rewrites* to a term t' using such an equation if there is a subterm $t|_p$ of t at a given position[14] p of t such that l matches $t|_p$ via a well-sorted substitution[15] σ and t' is obtained from t by replacing the subterm $t|_p \equiv \sigma(l)$ with the term $\sigma(r)$. In addition, if the equation has a condition *cond*, the substitution σ must make the condition provably true according to the equations and memberships in E, which are assumed to be Church-Rosser and terminating and are used also from left to right to try to simplify the condition. Note that, in general, the variables instantiated by σ must contain both those in the lefthand side, and those in the condition (which are incrementally matched using the matching equations).

We denote this step of *equational simplification* by $t \to_E t'$, where the possible equations for rewriting are chosen in the set E. The reflexive and transitive closure of the relation $\to_E$ is denoted $\to_E^*$.

In many texts, equational simplification is also called *(equational) rewriting* but, since in Maude we have two very different types of rewriting, rewriting with *equations* in functional modules, and rewriting with *rules* in system modules, each with a completely different semantics, to avoid confusion we favor the terminology of equational simplification for the process of rewriting with equations.

A set of equations E is *confluent* when any two rewritings of a term can always be unified by further rewriting: 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 $t_2 \to_E^* t'$. This is summarized in Figure 4.1.

A set of equations E is *terminating* when there is no infinite sequence of rewriting steps

$$t_0 \to_E t_1 \to_E t_2 \to_E \dots$$

[13]Some authors would instead say that u matches t.

[14]We can represent a term t as a tree, and use strings of numbers to identify positions p in the tree, thus identifying subterms $t|_p$. For example, for $t = f(g(a), h(b))$, we have $t|_{nil} = t$, $t|_1 = g(a)$, $t|_{11} = a$, $t|_2 = h(b)$, and $t|_{21} = b$.

[15]Note that if a variable x has a sort s instead of a kind, well sortedness of σ means that $\sigma(x)$ must provably have sort s (or lower) according to the equations E.

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

In membership equational theories a third important property is *sort decreasingness*. Intuitively, this means that, assuming E is confluent and terminating, the canonical form $t\downarrow_E$ of a term t by the equations E should have the *least sort possible* among the sorts of all the terms equivalent to it by the equations E; and it should be possible to compute this least sort from the canonical form itself, using only the operator declarations and the memberships. By a *Church-Rosser and terminating* theory (Σ, E) we precisely mean one that is confluent, terminating, and sort-decreasing. For a more detailed treatment of these properties, we refer the reader to the paper [17].

Since Maude functional modules have an initial algebra semantics, we are primarily interested in *ground* terms. Therefore, we can relax the above Church-Rosser and termination requirements by requiring that they just hold for ground terms, without losing the desired coincidence between the mathematical and operational semantics. In this way, we obtain notions of *ground* Church-Rosser, terminating, confluent, etc. specifications. In practice, some perfectly reasonable Maude functional modules are ground confluent, but fail to be confluent. This however is not a problem, since ground confluence (together with ground termination) is just what is needed to ensure uniqueness of canonical forms. Indeed, under the ground Church-Rosser and termination assumptions, it is easy to prove that we have the desired isomorphism

$$T_{\Sigma/E} \cong Can_{\Sigma/E}$$

ensuring the coincidence between the *mathematical semantics* of (Σ, E) provided by the initial algebra $T_{\Sigma/E}$, and its *operational semantics* by equational simplification provided by the algebra $Can_{\Sigma/E}$ of canonical forms.

Equational specifications (Σ, E) in Maude functional modules (and in the equational part of system modules), are assumed to be ground Church-Rosser and terminating up to the context-sensitive strategy specified by the evaluation strategies declared for the operators in Σ (see Section 4.4.7). More precisely, we can view the information about operator evaluation strategies as a function μ that assigns to each operator $f \in \Sigma$ with n argument sorts a string of numbers indicating the argument positions to be evaluated and ended with a 0 (that is, the information given in the operator's `strat` attribute, or, if no such information is given, the string $1 \ldots n\, 0$). This then defines a more restricted rewrite relation $\rightarrow_E^\mu$ where we can only rewrite in subterms in positions that can be evaluated according to μ. If the relation $\rightarrow_E^\mu$ is (ground) confluent, we call the specification (ground) *μ-confluent*; similarly, if $\rightarrow_E^\mu$ is (ground) terminating, we call it (ground) *μ-terminating*. We define the concepts of (ground) *μ-sort-decreasing* and (ground) *μ-Church-Rosser* in the same way. When we talk about the specification being "ground Church-Rosser and terminating up to the context-sensitive strategy specified by the evaluation strategies," we exactly mean that it is ground μ-Church-Rosser and ground μ-terminating. Of course, when no such strategies are declared, this specializes to the usual notions of ground Church-Rosser and ground terminating. Under the ground μ-Church-Rosser and ground μ-terminating assumptions, the μ-canonical forms define a canonical term algebra $Can^\mu_{\Sigma/E}$ (see [81]), which provides a perfect mathematical model for the module's *operational semantics*, since its elements are the *values* that the user gets when evaluating expressions in such a module. The question then arises: how is this model related to the module's *mathematical semantics*? In general, the quotient map $t \mapsto [t]_E$ sending each μ-canonical form to its

E-equivalence class is a *surjective* homomorphism

$$q : Can^{\mu}_{\Sigma/E} \longrightarrow T_{\Sigma/E},$$

but not necessarily an isomorphism. If q fails to be an isomorphism, this means that μ-rewriting is a *sound* deductive method for proving E-equalities, but it is *incomplete*. Therefore we call the specification *μ-semantically complete* iff q is an isomorphism. μ-semantic completeness therefore expresses the complete agreement between the mathematical and operational semantics of the module. Specifications where evaluation strategies are used mainly for efficiency and/or termination purposes, that is, those where the execution becomes more efficient by avoiding wasteful computation in unnecessary parts of the term and/or that would not terminate without the strategy restrictions are typically μ-semantically complete. Instead, specifications such as the sieve of Eratosthenes in Section 4.4.7, where the main intent is to compute with infinite data structures in a lazy way, are typically μ-semantically incomplete. Not all is lost in this second case: we still have a good mathematical model associated to our specification, namely, $Can^{\mu}_{\Sigma/E}$, but this is a more concrete model than $T_{\Sigma/E}$, that is, one in which fewer elements are identified.

What are the appropriate notions when we have a theory of the form $(\Sigma, E \cup A)$? Then matching must be defined *modulo the equational axioms* A, and all the above concepts, including those for μ-rewriting, must be generalized to equational simplification, confluence, and termination *modulo* A. We discuss this in more detail in Section 4.8 below. See also [81] for a detailed treatment of μ-rewriting and μ-semantic completeness modulo axioms A.

As already mentioned, the operational semantics of functional modules is *equational simplification*, that is, equational rewriting of terms until a canonical form is obtained in the sense explained above. Notice that the system does not check the ground confluence and termination properties: they are left to the user's responsibility. However, in some cases it is possible to check these properties with Maude's Church-Rosser checker and termination tools [30, 54, 47, 46]. Similar checkings are also possible for functional modules with evaluation strategies; for example, the Maude's MTT termination tool can check μ-termination (also called context-sensitive termination [94]). Moreover, although the relations between the standard Church-Rosser property and the μ-Church-Rosser property are somewhat subtle [93, 95], the work in [81] shows how one can use standard tools in conjunction with Maude's Sufficient Completeness Checker [83] to check both the μ-Church-Rosser property and μ-semantic completeness. See Section 11.4 for some examples of the use of these formal tools.

4.8 More on matching and simplification 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 4.7 above).

Given an equational theory $A = \cup_i A_{f_i}$ corresponding to all the attributes declared in different binary operators, 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 Chapter 5) modulo the axioms A.

Note, however, that for operators 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, assuming a binary associative operator `f` and constants `a`, `b`, `c`, and `d` of the appropriate sort, the term t = `f(a,b)` does not match the term u = `f(a, f(b, f(c, d)))`, that is, there is no substitution making both terms equal modulo associativity; however, because of associativity of `f`, u is equivalent to `f(f(a, b), f(c, d))` and then t trivially matches the first subterm. This becomes easier to see using mixfix notation; if f =`_._`, then t = `a . b` and u = `a . b . c . d`, and we clearly see that t matches a fragment of u. 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 operator 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 equivalence classes modulo A. For more technical details on rewriting modulo a set of axioms, see, e.g., [40].

Matching with extension for an associative operator 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 equation `a . b = e` 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 operator f include the associativity axiom, then a subject term u with f as its top operator is internally represented (but this representation can also be externally used, see Section 3.9.3) as the *flattened term* $f(u_1, \ldots, u_n)$, with the $u_1, \ldots, u_n$ having top operators 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 the term $f(u_1, \ldots, u_n)$ 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 $2 \leq h \leq n$.

The concepts of positions in a term and depth of a term, that are important in many situations, refer to this flattened form. The compact notation for terms constructed with operators having the `iter` attribute (Section 4.4.2) is also considered a form of flattened notation, so that, for the purpose of calculating term depth, if the top level is at level 0, then the occurrence of `X:Foo` in `f^3(X:Foo)` is at level 1, not level 3.

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, 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,

$$\begin{aligned} \texttt{d . a} &\rightarrow \texttt{d . b . a} \\ &\rightarrow \texttt{d . b . b . a} \\ &\vdots \\ &\rightarrow \texttt{d . b . b . } \cdots \texttt{ . b . a} \\ &\vdots \end{aligned}$$

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 operator may, in the presence of an additional identity attribute, match a term not involving at all that operator. Thus, for the above equation, we have also the nonterminating chain of rewriting steps

$$\begin{aligned} \texttt{a} &\rightarrow \texttt{b . a} \\ &\rightarrow \texttt{b . b . a} \\ &\vdots \\ &\rightarrow \texttt{b . b . } \cdots \texttt{ . b . a} \\ &\vdots \end{aligned}$$

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

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

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

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

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 item 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` (because the lefthand side `a` is not a `_._`-term), 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 operators, 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) [metadata "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 (see also Section 23.3) for just this purpose:

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

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

```
Maude> xmatch X . (Y + Z) <=? a . b . (c + d + e) .
xmatch in XMATCH-TEST : X . Z + Y <=? a . b . c + d + e .
Decision time: 0ms cpu (0ms real)

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 the reason why the subterm `Y + Z` of the lefthand side should match the entire maximal `_+_`-subterm of the subject term, and not just some `_+_`-subterm.

For operators with the `iter` attribute, the situation with matching is analogous to the `assoc` theory, so that proper subterms of say `f^3(X:Foo)`, such as `f^2(X:Foo)` and `f(X:Foo)`, can also be matched by means of extension.

4.9 The `reduce`, `match`, `trace`, and `show` commands

Here we assemble the whole module for the `NUMBERS` running example to illustrate some of the basic commands for interacting with Maude. See Chapter 23 for full details about these and other Maude commands.

Notice that, since the result of the `_in_` predicate is a Boolean value, we import the predefined module `BOOL` (see Section 7.1) by means of a `protecting` declaration (described in Section 6.1.1).

```
fmod NUMBERS is
  protecting BOOL .

  sort Zero .
  sorts Nat NzNat .
  subsort Zero NzNat < Nat .
  op zero : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor] .
```

```
  op sd : Nat Nat -> Nat .
  ops _+_ _*_ : Nat Nat -> Nat [assoc comm] .
  op _+_ : NzNat Nat -> NzNat [ditto] .
  op _*_ : NzNat NzNat -> NzNat [ditto] .
  op p : NzNat -> Nat .

  vars I N M : Nat .
  eq N + zero = N .
  eq N + s M = s (N + M) .
  eq sd(N, N) = zero .
  eq sd(N, zero) = N .
  eq sd(zero, N) = N .
  eq sd(s N, s M) = sd(N, M) .
  eq N * zero = zero .
  eq N * s M = (N * M) + N .
  eq p(s N) = N [label partial-predecessor] .

  eq (N + M) * I = (N * I) + (M * I)
    [nonexec metadata "distributive law"] .

  sort Nat3 .
  ops 0 1 2 : -> Nat3 [ctor] .
  op _+_ : Nat3 Nat3 -> Nat3 [comm] .
  vars N3 : Nat3 .
  eq N3 + 0 = N3 .
  eq 1 + 1 = 2 .
  eq 1 + 2 = 0 .
  eq 2 + 2 = 1 .

  sort NatSeq .
  subsort Nat < NatSeq .
  op nil : -> NatSeq .
  op __ : NatSeq NatSeq -> NatSeq [assoc id: nil] .

  sort NatSet .
  subsort Nat < NatSet .
  op empty : -> NatSet .
  op _;_ : NatSet NatSet -> NatSet [assoc comm id: empty] .
  eq N ; N = N [label natset-idem] .

  op _in_ : Nat NatSet -> Bool .
  var NS : NatSet .
  eq N in N ; NS = true .
  eq N in NS = false [owise] .
endfm
```

First, we evaluate some expressions using the `reduce` command. Maude repeats the command filling in any omitted optional information. Then statistics about the execution are printed.[16] Finally, the result is printed, prefaced by its least sort.

The first two examples evaluate the sum of three ones in `Nat` and in `Nat3`.

```
Maude> red s zero + s zero + s zero .
```

[16]The cpu and real time information is not printed if the user has made use of the `set show timing off` command (see Section 23.11).

```
reduce in NUMBERS : s zero + s zero + s zero .
rewrites: 4 in 0ms cpu (0ms real) (~ rews/sec)
result NzNat: s s s zero

Maude> red 1 + (1 + 1) .
reduce in NUMBERS : 1 + (1 + 1) .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result Nat3: 0
```

The next example illustrates the effect of the idempotency equation for sets of natural numbers.

```
Maude> red zero ; s zero ; zero ; s zero .
reduce in NUMBERS : zero ; s zero ; zero ; s zero .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result NatSet: zero ; s zero
```

Finally we convince ourselves that the `owise` attribute works.

```
Maude> red zero in s zero ; zero ; s s zero .
reduce in NUMBERS : zero in s zero ; zero ; s s zero .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Bool: true

Maude> red zero in s zero ; s s zero .
reduce in NUMBERS : zero in s zero ; s s zero .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Bool: false
```

The commands `xmatch` and `match` operate in the same way, unless the subject term has an operator that needs extension on top, in which case it can match proper subterms in the same theory layer, as required for rewriting modulo that theory. The `xmatch` command was illustrated in Section 4.8. Here we compare `match` and `xmatch` on a pattern that splits a sequence of natural numbers into two parts. To be safe, we ask for at most five matches, but in fact there are only four.

```
Maude> match [5] NS0:NatSeq NS1:NatSeq <=? zero zero zero .
match [5] in NUMBERS : NS0:NatSeq NS1:NatSeq <=? zero zero zero .
Decision time: 0ms cpu (0ms real)

Solution 1
NS0:NatSeq --> nil
NS1:NatSeq --> zero zero zero

Solution 2
NS0:NatSeq --> zero
NS1:NatSeq --> zero zero

Solution 3
NS0:NatSeq --> zero zero
NS1:NatSeq --> zero

Solution 4
NS0:NatSeq --> zero zero zero
NS1:NatSeq --> nil
```

Using the `xmatch` command for the same pattern and term, we see that in addition to the whole term matches, Maude also reports matches within the subterm `zero zero`. In fact, there are two occurrences of this subterm. We only show five of the matches.

```
Maude> xmatch [5] NS0:NatSeq NS1:NatSeq <=? zero zero zero .
xmatch [5] in NUMBERS : NS0:NatSeq NS1:NatSeq <=? zero zero zero .
Decision time: 0ms cpu (7ms real)

Solution 1
Matched portion = zero zero
NS0:NatSeq --> nil
NS1:NatSeq --> zero zero

Solution 2
Matched portion = zero zero
NS0:NatSeq --> zero
NS1:NatSeq --> zero

Solution 3
Matched portion = zero zero
NS0:NatSeq --> zero zero
NS1:NatSeq --> nil

Solution 4
Matched portion = (whole)
NS0:NatSeq --> nil
NS1:NatSeq --> zero zero zero

Solution 5
Matched portion = (whole)
NS0:NatSeq --> zero
NS1:NatSeq --> zero zero
```

Let us consider now a small example using the `trace` command. We turn on selective tracing and choose to trace only uses of the equation labeled `partial-predecessor`.

```
Maude> set trace on .
Maude> set trace select on .
Maude> trace select partial-predecessor .

Maude> red s s p(s zero) + s p(s zero) .
reduce in NUMBERS : s s p(s zero) + s p(s zero) .
********** equation
eq p(s N) = N [label partial-predecessor] .
N --> zero
p(s zero)
--->
zero
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result NzNat: s s s zero
```

Note that Maude only reports one use of this equation, despite the fact that there are two occurrences in the term. This is because, when performing equational simplification, occurrences of the same subterm are internally shared[17] and hence there is only one occurrence of

[17]However, this sharing—i.e., treating the term as a dag instead of as a tree—is not done in a *maximal* way, so

the subterm `p(s zero)` in the internal representation.

We can ask Maude to show the module `FIBONACCI` (assuming it has been loaded).

```
Maude> show module FIBONACCI .
fmod FIBONACCI is
  protecting NAT .
  op fibo : Nat -> Nat [memo] .
  var N : Nat .
  eq fibo(0) = 0 .
  eq fibo(1) = 1 .
  eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

The `show sorts` command shows all the sorts declared and for each sort its sub- and supersorts.

```
Maude> show sorts NUMBERS .
sort Bool .
sort Zero . subsorts Zero < Nat NatSet NatSeq .
sort Nat . subsorts NzNat Zero < Nat < NatSet NatSeq .
sort NzNat . subsorts NzNat < Nat NatSet NatSeq .
sort Nat3 .
sort NatSeq . subsorts NzNat Zero Nat < NatSeq .
sort NatSet . subsorts NzNat Zero Nat < NatSet .
```

The `show components` command shows the connected components (kinds) in the sort partial order.

```
Maude> show components NUMBERS .
[Bool]:
       1      Bool

[NatSeq, NatSet]:
       1      NatSeq
       2      NatSet
       3      Nat
       4      Zero
       5      NzNat

[Nat3] (error free):
       1      Nat3
```

Note that the name of the kind corresponding to the connected component containing the natural numbers contains the names of two sorts. These are the maximal sorts in the component. The `(error free)` comment about the sort `Nat3` means that all terms of kind `[Nat3]` are in fact of sort `Nat3`.

---

that all subterms that can be shared are; instead, term sharing is itself introduced incrementally by equational simplification, since Maude analyzes righthand sides of equations to identify its shared subterms. As explained by Eker in [58], in the presence of operator evaluation strategies (Section 4.4.7) term sharing has to be done carefully. Furthermore, when rewriting is performed with a rule in a system module (see Chapter 5), rather than with an equation, Maude will incrementally *unshare* those parts of the subject term needed to ensure that *all* possible rewrite with rules are considered. This is because rules in system modules need not be confluent. As a consequence, two identical subterms could be rewritten in totally different ways; but this of course would be prevented if they were to be shared.

Chapter 5

System Modules

A Maude system module specifies a *rewrite theory*. A rewrite theory has sorts, kinds, and operators (perhaps with frozen arguments), and can have three types of statements: equations, memberships, and rules, all of which can be conditional. Therefore, any rewrite theory has an underlying equational theory, containing the equations and memberships, *plus* the rules. What is the intuitive meaning of such rules? Computationally, they specify *local concurrent transitions* that can take place in a system if the pattern in the rule's lefthand side matches a fragment of the system state and the rule's condition is satisfied. In that case, the transition specified by the rule can take place, and the matched fragment of the state is transformed into the corresponding instance of the righthand side. Logically, that is, when we use rewriting logic as a logical framework to represent other logics as explained in Section 1.4, a rule specifies a *logical inference rule*, and rewriting steps therefore represent inference steps.

As was mentioned in Section 3.2, a system module is declared in Maude using the keywords

`mod` ⟨*ModuleName*⟩ `is` ⟨*DeclarationsAndStatements*⟩ `endm`

As for functional modules the first bit of information in the specification is the module's name, which must be an identifier. For example,

```
mod VENDING-MACHINE is
  ...
endm
```

where the dots stand for all the declarations and statements in the module, which can be:

1. module importations,
2. sort and subsort declarations,
3. operator declarations,
4. variable declarations,
5. equation and membership statements, and
6. rule statements.

Since declarations (1)–(4) and equational statements (5) are exactly as for functional modules, all we have left to explain is how rules (conditional or not) are declared. As for equation and membership statements, rules can be declared with any of the attributes `label`, `metadata`, `nonexec`, and `print` (see Section 4.5). However, the `owise` attribute can only be used with equations.

5.1 Unconditional rules

Mathematically, an unconditional rewrite rule has the form $l : t \to t'$, where t, t' are terms of the same kind, which may contain variables, and l is the label of the rule. Intuitively, a rule describes a *local concurrent transition* in a system: anywhere in the distributed state where a substitution instance $\sigma(t)$ of the lefthand side t is found, a local transition of that state fragment to the new local state $\sigma(t')$ can take place. And if many instances of the same or of several rules can be matched in different nonoverlapping parts of the distributed state, then all of them can fire concurrently.

An unconditional rule is introduced in Maude with the following syntax:

`rl [`⟨*Label*⟩`] :` ⟨*Term-1*⟩ `=>` ⟨*Term-2*⟩ `[`⟨*StatementAttributes*⟩`] .`

As explained in Section 4.5.1, a label can alternatively be declared as a statement attribute; also, Maude allows declaration of *unlabeled rules*. In these two cases, the part "`[`⟨*Label*⟩`] :`" is omitted.

As a first example of a system module we consider the following specification of a vending machine which dispenses apples and cakes. The module `VENDING-MACHINE-SIGNATURE` is the underlying functional module. This module is imported by the system module `VENDING-MACHINE`, which then adds the rules for operating the machine. Although not necessary, in addition to making the underlying functional module explicit, such splitting of modules can be useful in organizing a large specification, where a functional part may be shared by several system modules; see Chapter 6 for a discussion on module importation.

The constants `$` and `q` represent coins of one dollar and one quarter, respectively, while the constants `a` and `c` represent apples and cakes, respectively.

```
fmod VENDING-MACHINE-SIGNATURE is
  sorts Coin Item Marking .
  subsorts Coin Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: null] .
  op null : -> Marking .
  op $ : -> Coin [format (r! o)] .
  op q : -> Coin [format (r! o)] .
  op a : -> Item [format (b! o)] .
  op c : -> Item [format (b! o)] .
endfm
```

The `format` declaration for each constant (see Section 4.4.5) is used to print the constants using different colors, so that coins can easily be separated from items in a given marking.

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change] : q q q q => $ .
endm
```

This module specifies a concurrent machine to buy cakes and apples with dollars and quarters. A cake costs a dollar, and an apple three quarters. We can insert dollars and quarters in the machine, although due to an unfortunate design, the machine only accepts buying cakes and apples with dollars. When the user buys an apple the machine takes a dollar and returns a quarter. To alleviate in part this problem, the machine can change four quarters into a dollar.

The machine is *concurrent*, because we can *push several buttons at once* (that is, we can apply several rules at once), provided enough resources exist in the corresponding slots, called *places*. For example, if we have one dollar in the `$` place and four quarters in the `q` place, we can *simultaneously* push the `buy-a` and `change` buttons, and the machine returns, also simultaneously, one dollar in `$`, one apple in `a`, and one quarter in `q`.

Note that, since the Maude interpreter is sequential, the above concurrent transitions in the `VENDING-MACHINE` module are simulated by corresponding *interleavings* of sequential rewriting steps. In a socket-based concurrent implementation, it is possible to execute concurrently many rewriting steps for a wide range of system modules.[1]

We might have tried a simpler alternative, namely, using the rule `null => q` instead of the `add-q` rule. However, this would not work. Instead, we have to write `M => M q` with `M` a variable of sort `Marking`. The reason is that the constant `null` is not a `__`-subterm of any marking except itself, and therefore it would be impossible to apply the rule `null => q` with extension (see Section 4.8).

5.2 Conditional rules

Conditional rewrite rules can have very general conditions involving equations, memberships, and other rewrites; that is, in their mathematical notation they can be of the form

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

with no restriction on which new variables may appear in the righthand side or the condition. There is no need for the condition listing first equations, then memberships, and then rewrites: this is just a notational abbreviation, since they can be listed in any order. However, in Maude, conditions are evaluated from left to right, and therefore the order in which they appear, although mathematically inessential, is very important operationally (see Section 5.3).

In their Maude representation, conditional rules are declared with syntax

```
crl [⟨Label⟩] : ⟨Term-1⟩ => ⟨Term-2⟩
  if ⟨Condition-1⟩ /\ ... /\ ⟨Condition-k⟩
  [⟨StatementAttributes⟩] .
```

where the rule's label can instead be declared as a statement attribute, or can be omitted altogether. In either of these two alternatives, the square brackets enclosing the label and the colon are then omitted.

As in conditional equations, the condition can consist of a single statement or can be a conjunction formed with the associative connective `/\`. But now conditions are more general, since in addition to equations and memberships they can also contain rewrite expressions, for which the concrete syntax `t => t'` is used. Furthermore, equations, memberships, and rewrites can be intermixed in any order. As for functional modules, some of the equations in conditions can be either matching equations or abbreviated Boolean equations.

We can illustrate the usefulness of rewrite expressions in rule conditions by presenting a small fragment of a Maude operational semantics for Milner's CCS language given in [136]:

```
sorts Label Act Process ActProcess .
subsorts Qid < Label < Act .
subsort Process < ActProcess .
```

[1] See [29, Chapter 16] for an interesting example of this kind: a concurrent implementation of a mobile language entirely programmed in Maude using sockets as external objects in the way explained in Section 9.3.

```
op ~_ : Label -> Label .
op tau : -> Act .
op {_}_ : Act ActProcess -> ActProcess [frozen] .
op _|_ : Process Process -> Process [frozen assoc comm] .

vars P P' Q Q' : Process .
var  L : Label .

crl [par] : P | Q => {tau} (P' | Q')
  if P => {L} P' /\ Q => {~ L} Q' .
```

The conditional rule `par` expresses the synchronized transition of two processes composed in parallel. The condition of the rule states that the synchronized transition can take place if one process can perform an action named `L` and the other can perform the complementary action named `~ L`. In this representation of CCS, the action performed is remembered by the resulting expression, which is a term of sort `ActProcess`.

Note the use of the `frozen` attribute in some of the operators (see Section 4.4.9).

5.3 Admissible system modules

The same way that equations or memberships expressed in their fullest possible generality cannot be executed by the Maude engine except in a controlled way at the metalevel, conditional rewrite rules in their fullest generality cannot be executed either, except with a strategy at the metalevel. Nonexecutable rules should be identified by giving them the `nonexec` attribute.

As for functional modules, the question now becomes: what are the executability requirements on the executable statements (i.e., those without the `nonexec` attribute) of a system module? It turns out that a quite general class of system modules, called *admissible modules*, are executable by Maude's default interpreter using the `rewrite`, `frewrite`, and `search` commands, that will be introduced and illustrated in Section 5.4 and are further explained in Sections 23.2 and 23.4.

The admissibility requirements for the module's equations and memberships are exactly as for functional modules; they were explained in Section 4.6 and are further discussed below. Two more requirements are needed:

- each executable conditional rule should be *admissible*, and
- the rules should be *coherent* relative to the equations, as has already been mentioned in the introduction.

We explain each of these requirements below.

Given a system module M, a conditional[2] rule of the form

$$l : t \to t' \;\; \textit{if} \;\; C_1 \wedge \ldots \wedge C_n$$

such that it does not have the `nonexec` attribute is called *admissible* if it satisfies the exact analogues of the admissibility requirements 1–3 in Section 4.6 for conditional equations, plus the additional requirement

[2]For the purposes of this discussion, we view unconditional rules as a special case of conditional rules. The general admissibility requirement specializes then to a very easy requirement for an unconditional rule $t \to t'$, namely, that each variable of t' must appear in t.

4. If C_i is a rewrite $u_i \to u_i'$, then

$$vars(u_i) \subseteq vars(t) \cup \bigcup_{j=1}^{i-1} vars(C_j),$$

and furthermore u_i' is an $\mathcal{E}(M)$-pattern (for the notion of pattern see Section 4.3) for $\mathcal{E}(M)$ the equational theory underlying the module M.

Operationally, we try to satisfy such a rewrite condition by reducing the substitution instance $\sigma(u_i)$ to its canonical form v_i with respect to the equations, and then trying to find a rewrite proof $v_i \to w_i$ (perhaps after many steps of rewriting) with w_i in canonical form with respect to the equations and such that w_i is a substitution instance of u_i'. Search for such a w_i is performed by the Maude engine using a breadth-first strategy.

As for functional modules, when executing an admissible conditional rule in a system module, the satisfaction of all its conditions is attempted sequentially from left to right; but notice that now, besides the fact that many matches for the equational conditions may be possible due to the presence of equational attributes, we also have to deal with the fact that solving rewrite conditions requires *search*, including searching for new solutions when previous ones fail to satisfy subsequent conditions.

We now explain the *coherence* requirement. 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 equational theory is undecidable.

The equations and memberships specified in a system module M are divided into a set A of axioms corresponding to equational attributes such as associativity, commutativity, idempotency, and (left-, right- or two-sided) identity declared for different operators in the module (see Section 4.4.1), for which matching algorithms exist in the Maude implementation, and a set E of equations and memberships specified in the ordinary way. As already mentioned, for M to be executable, the set of executable statements in E must be Church-Rosser and terminating *modulo* A, or at least ground Church-Rosser and terminating modulo A; that is, we require that the equational part must be equivalent to an executable functional module.

Moreover, we require that the rules R in the module are *coherent* [138] with respect to the equations E modulo A, or at least *ground coherent*. Coherence means that, given a term t, for each one-step rewrite of it with some rule in R modulo the axioms A to some term t', which we denote $t \longrightarrow^1_{R/A} t'$, if u is the canonical term we obtain by rewriting t with the equations and memberships in E to canonical form modulo A, denoted $t \longrightarrow^!_{E/A} u$, then there is a one-step rewrite of u with some rule in R modulo A, $u \longrightarrow^1_{R/A} u'$, such that $t' =_{E \cup A} u'$, which by the Church-Rosser and termination properties of E modulo A is equivalent to t' and u' having the same canonical form modulo A by E. This requirement is described graphically in Figure 5.1.

Ground coherence is a weaker requirement: we require the exact same diagram to exist only for *ground* terms, and E only needs to be ground Church-Rosser and terminating modulo A.

As explained in [138] (for the free case and for coherence modulo associativity and commutativity), for unconditional rules R, coherence can be checked by checking "critical pairs" between rules R and equations E, and showing that the corresponding instance of the coherence diagram can be filled in for all such pairs. That is, we have to look for appropriate overlaps between lefthand sides of rules and equations using an A-unification algorithm, generate the corresponding critical pairs, and check their coherence. In the case of ground coherence, it is not necessary that the critical pairs can be filled in: it is enough to show that each *ground*

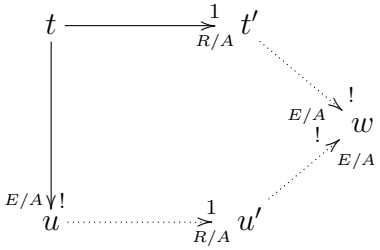


Figure 5.1: Coherence diagram

instance of such pairs can be filled in. See Section 7.8 of [29] for an example of a system module that is not coherent, a discussion of the critical pairs involved, and a method to make the specification coherent. See also Section 13.4 of [29] for an example of how coherence can be checked by critical pair analysis. Similarly, for ground coherence and how to check it, see the example in Section 11.4.

Why is coherence so important? What does it mean intuitively? Rewriting modulo an equational theory $E \cup A$, which is what a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$ really does, is in general undecidable. The undecidability has to do with the fact that we may need to search an entire $E \cup A$-equivalence class before we can know if a class representative can be rewritten with R, that is, if the $E \cup A$-equivalence class can be rewritten. Coherence makes the problem decidable: all we need to do is to reduce the term to its canonical form by E modulo A, and then rewrite with R such a canonical form. In a sense, coherence reduces rewriting with R modulo $E \cup A$ to rewriting with E and R modulo A, which is decidable, because we assume we have an A-matching algorithm.

Could we miss any rewrites that way? Coherence guarantees to us that we could not, since any rewrite of a term t with R must also be possible with t's canonical form. Maude implicitly assumes this coherence property. For example, the `rewrite` command will at each step first reduce the term to canonical form with E modulo A, and then perform a rewrite step with R in a rule-fair manner. The `frewrite` command uses a somewhat different rewrite strategy to ensure both local fairness and rule fairness, but assumes the same coherence (or ground coherence) property (see Section 23.2 and examples in Section 5.4 below).

A last point about the execution of system modules regards *frozen* argument positions in operators (see Section 4.4.9). This poses a general constraint on any rewriting strategy whatsoever, including those directly supported by Maude for the `rewrite` and `frewrite` commands (see Section 5.4). The general constraint is that *rewriting will never happen below one of the frozen argument positions* in an operator. That is, even though many rewritings may be possible and there can be a large amount of nondeterminism (so that different rewriting strategies may lead to quite different results) rewriting under frozen arguments is *always forbidden*. In fact, this does not only belong to the module's operational semantics, but also to the latest initial model semantics for rewrite theories developed in [19]; we give a brief informal summary of this semantics below.

Mathematically, a system module, when "flattened" with its imported submodules, exactly specifies a (generalized) *rewrite theory* in the sense of [19], that is, a four-tuple

$$\mathcal{R} = (\Sigma, E \cup A, \phi, R),$$

where $(\Sigma, E \cup A)$ is the membership equational theory specified by the signature, equational attributes, and equation and membership statements in the module (just as in the case of

functional modules); ϕ is a function, assigning to each operator in Σ the set of natural numbers corresponding to its frozen arguments (the empty set when no argument is frozen); and R is the collection of (possibly conditional) rewrite rules specified in the module and its submodules.

Intuitively, such a rewrite theory specifies a *concurrent system*. The equational theory $(\Sigma, E \cup A)$ specifies the "statics" of the system, that is, the algebraic structure of the set[3] of states, which is specified by the initial algebra $T_{\Sigma/E\cup A}$. The rules R and the freezing information ϕ specify the concurrent system's "dynamics," that is, the possible concurrent transitions that the system can perform. In rewriting logic, such, possibly complex, concurrent transitions exactly correspond to *rewrite proofs*; but since several rewrite proofs can indeed correspond to the *same* concurrent computation (describing, for example, different semantically equivalent interleavings), rewriting logic has an equational theory of *proof equivalence* [102, 19].

The *initial model* $\mathcal{T}_{\mathcal{R}}$ of the rewrite theory $\mathcal{R}$ associates to each kind k a labeled transition system (in fact, a *category*) whose set of states is $T_{\Sigma/E\cup A,k}$, and whose labeled transitions have the form $[\alpha] : [t] \to [t']$, with $[t], [t'] \in T_{\Sigma/E\cup A,k}$, and with $[\alpha]$ an equivalence class of rewrite proofs modulo the equational theory of proof equivalence. Indeed what the different $[\alpha]$ represent are the different "truly concurrent" computations of the system specified by $\mathcal{R}$.

5.4 The `rewrite`, `frewrite`, and `search` commands

Now we illustrate the use of the Maude commands available for system modules. Recall the vending machine example:

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change]: q q q q => $ .
endm
```

In addition to the `show` commands discussed in Section 4.9, there is an additional `show rls` command for system modules to show the rules of a module. For example, showing the sorts and the rules of the `VENDING-MACHINE` module we get:

```
Maude> show sorts VENDING-MACHINE .
sort Bool .
sort Coin .    subsort Coin < Marking .
sort Item .    subsort Item < Marking .
sort Marking . subsorts Item Coin < Marking .

Maude> show rls VENDING-MACHINE .
rl M => q M [label add-q] .
rl M => $ M [label add-$] .
rl $ => c [label buy-c] .
rl $ => q a [label buy-a] .
rl q q q q => $ [label change] .
```

[3] More precisely, each kind k in Σ corresponds to a different choice for a set of states, namely the set $T_{\Sigma/E\cup A,k}$.

5.4.1 The `rewrite` command

We can use the `rewrite` command (abbreviated `rew`) to explore the behavior of different initial markings. The bracketed number between the command and the term to be rewritten provides an upper bound for the number of rule applications that are allowed.

```
Maude> rew [1] in VENDING-MACHINE : $ $ q q .
rewrite [1] in VENDING-MACHINE : $ $ q q .
rewrites: 1 in 0ms cpu (9ms real) (~ rews/sec)
result Marking: $ $ q q q

Maude> rew [2] $ $ q q .
rewrite [2] in VENDING-MACHINE : $ $ q q .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ q q q

Maude> rew [3] $ $ q q .
rewrite [3] in VENDING-MACHINE : $ $ q q .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ q q q q

Maude> rew [4] $ $ q q .
rewrite [4] in VENDING-MACHINE : $ $ q q .
rewrites: 4 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ $ q q q q

Maude> rew [5] $ $ q q .
rewrite [5] in VENDING-MACHINE : $ $ q q .
rewrites: 5 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ $ $

Maude> rew [6] $ $ q q .
rewrite [6] in VENDING-MACHINE : $ $ q q .
rewrites: 6 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ $ $ q

Maude> rew [200] $ $ q q .
rewrite [200] in VENDING-MACHINE : $ $ q q .
rewrites: 200 in 10ms cpu (10ms real) (20000 rews/sec)
result Marking: $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
    $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
    $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
    $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ q q q
```

Executing one rewrite starting with two dollars and two quarters, Maude chooses to apply the `add-q` rule. If we allow two rewrites Maude applies `add-q` and then `add-$`. The third rule to be applied is `add-q` again; then, `add-$`. It goes on applying `add-q` and `add-$` until the rule `change` can be applied. The top-down rule-fair `rewrite` strategy keeps trying to apply rules on the top operator (`__` in this case) in a fair way. The rules applicable at the top are `add-q`, `add-$`, and `change`, which are tried in this order. Since the top operator is always the same one, no other rules are used. We can modify the rules `buy-c` and `buy-a` so that the lefthand side has an explicit top level `__` as follows:

```
mod VENDING-MACHINE-TOP is
  including VENDING-MACHINE-SIGNATURE .
```

```
  var M : Marking .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ M => c M .
  rl [buy-a] : $ M => a q M .
  rl [change]: q q q q => $ .
endm
```

Now starting with two dollars and two quarters, and executing increasing numbers of rewrites we see that Maude applies the rules `add-$`, `add-q`, `buy-c`, `buy-a`, and `change`.

```
Maude> rew [2] in VENDING-MACHINE-TOP : $ $ q q .
Advisory: "v.maude", line 18 (mod VENDING-MACHINE-TOP): collapse at
   top of $ M may cause it to match more than you expect.
Advisory: "v.maude", line 19 (mod VENDING-MACHINE-TOP): collapse at
   top of $ M may cause it to match more than you expect.
rewrite [2] in VENDING-MACHINE-TOP : $ $ q q .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ q q q

Maude> rew [3] $ $ q q .
rewrite [3] in VENDING-MACHINE-TOP : $ $ q q .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ q q q c

Maude> rew [4] $ $ q q .
rewrite [4] in VENDING-MACHINE-TOP : $ $ q q .
rewrites: 4 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ q q q q a c

Maude> rew [5] $ $ q q .
rewrite [5] in VENDING-MACHINE-TOP : $ $ q q .
rewrites: 5 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ a c
```

The advisory is about the modified rules for buying. Maude is letting us know that the pattern `$ M` will match a term not containing the top-level operator `__`, when `M` is instantiated to `null`. This is exactly what we want in this case, but it may not always be what the user intended, so Maude gives you a heads up; see Section 20.3.6 for more details.

Notice that rewriting in `VENDING-MACHINE` is not terminating. If we remove the rules for adding coins we obtain a terminating system and can explore vending behavior using unbounded rewriting. Let us consider the following module `SIMPLE-VENDING-MACHINE`.

```
mod SIMPLE-VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change]: q q q q => $ .
endm
```

For example, starting with two dollars and rewriting as much as possible we can get an apple, a cake and a quarter in change.

```
Maude> rew in SIMPLE-VENDING-MACHINE : $ $ .
rewrite in SIMPLE-VENDING-MACHINE : $ $ .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
```

```
result Marking: q a c
```

Starting with two dollars and three quarters and using only three rewrite rule applications we get an apple and a cake with a dollar left over.

```
Maude> rew [3] $ $ q q q .
rewrite [3] in SIMPLE-VENDING-MACHINE : $ $ q q q .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ a c
```

The command `continue` ⟨*Bound*⟩ (abbreviated `cont`) tells Maude to continue rewriting using at most ⟨*Bound*⟩ additional rule applications. For example, we can continue the last rewrite command (that performed three rewrites) for one more step to get an apple and two cakes:

```
Maude> cont 1 .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: a c c
```

5.4.2 The `frewrite` command

Let us see now what happens when we use another strategy for rewriting in the original `VENDING-MACHINE` module. The `frewrite` command (abbreviated `frew`) rewrites a term using a depth-first position-fair strategy that makes it possible for some rules to be applied that could be "starved" using the leftmost, outermost rule fair strategy of the `rewrite` command. The strategies for the `rewrite` and `frewrite` commands are described in detail in Section 23.2.

```
Maude> frew [2] in VENDING-MACHINE : $ $ q q .
frewrite [2] in VENDING-MACHINE : $ $ q q .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result (sort not calculated): ($ q) ($ $) q q

Maude> frew [12] $ $ q q .
frewrite [12] in VENDING-MACHINE : $ $ q q .
rewrites: 12 in 0ms cpu (0ms real) (~ rews/sec)
result (sort not calculated):
  c (q a) ($ q) ($ $) (q q) ($ q) (q q) q q
```

With two rewrites, one quarter and one dollar are added. With sufficiently many rewrites (twelve will do), a cake and an apple can be obtained.

In contrast to `rewrite`, that reduces the whole term using equations after each rule rewrite, `frewrite` only reduces the subterm rewritten (to preserve positions not yet visited). Thus, when rewriting stops, the term may not be fully reduced and hence Maude will not know the exact least sort of the term yet. This is the reason for the `(sort not calculated)` comment in place of a sort in the `result` line. In the case of a term with an associative and commutative top operator, the term may not be in its fully flattened form with canonical order of subterms. This accounts for the parentheses in the result term and the fact that the coins and items are not listed in order as they are in the result of a `rewrite`.

The top-down rule-fair strategy of the `rewrite` command can result in nontermination even though there is a terminating sequence of rewrites. As an example consider the following module:

```
mod BB-TEST is
  sort Expression .
```

```
  ops a b bingo : -> Expression .
  op f : Expression Expression -> Expression .

  rl a => b .
  rl b => a .
  rl f(b, b) => bingo .
endm
```

Giving the `rewrite` command with input term `f(a, a)` will result in the following looping computation:

```
f(a, a) => f(b, a) => f(a, a) => f(b, a) => f(a, a) => ...
```

This is because using the top-down rule-fair strategy of the `rewrite` command, the third rule always fails to match and never gets a chance to be applied. As already mentioned above, the `frewrite` command uses on the other hand a position-fair bottom-up strategy that makes it possible for other rules to be applied. As a consequence, some rewriting computations that could be nonterminating using the `rewrite` command become terminating with `frewrite`. For example, using the `frewrite` command in place of `rewrite` in the above example we get

```
Maude> frew in BB-TEST : f(a, a) .
frewrite in BB-TEST : f(a, a) .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Expression: bingo
```

5.4.3 The `search` command

The `rewrite` and `frewrite` commands each explore just one possible behavior (sequence of rewrites) of a system described by a set of rewrite rules and an initial state. The `search` command allows one to explore (following a breadth-first strategy) the reachable state space in different ways. Its syntax conforms to the following general scheme

`search [` n, m `] in` ⟨*ModId*⟩ `:` ⟨*Term-1*⟩ ⟨*SearchArrow*⟩ ⟨*Term-2*⟩
`such that` ⟨*Condition*⟩ .

where

- n is an optional argument providing a bound on the number of desired solutions;
- m is another optional argument stating the maximum depth of the search;
- the module ⟨*ModId*⟩ where the search takes place can be omitted;
- ⟨*Term-1*⟩ is the starting term;
- ⟨*Term-2*⟩ is the pattern that has to be reached;
- ⟨*SearchArrow*⟩ is an arrow indicating the form of the rewriting proof from ⟨*Term-1*⟩ until ⟨*Term-2*⟩:
  - `=>1` means a rewriting proof consisting of exactly one step,
  - `=>+` means a rewriting proof consisting of one or more steps,
  - `=>*` means a proof consisting of none, one, or more steps, and
  - `=>!` indicates that only canonical final states are allowed, that is, states that cannot be further rewritten; and

The one step arrow `=>1` is an abbreviation of the one-or-more steps arrow `=>+` with the depth bound m set to `1`.

- ⟨*Condition*⟩ states an optional property that has to be satisfied by the reached state; the syntactic form of the condition is the same as the one of conditions for conditional rules (see Section 5.2).

For example, for our *finite* vending machine, `SIMPLE-VENDING-MACHINE`, we can use the `search` command to answer the question: if I have a dollar and three quarters, can I get a cake and an apple? This is done by searching for states that match a corresponding pattern. In this example, we use the `=>!` symbol, meaning that we are searching for terminal states, that is, for states that cannot be further rewritten. Moreover, no bound in the number of solutions or in the depth of the search is needed.

```
Maude> search in SIMPLE-VENDING-MACHINE :
        $ q q q =>! a c M:Marking .
search in SIMPLE-VENDING-MACHINE : $ q q q =>! a c M:Marking .

Solution 1 (state 4)
states: 6 rewrites: 5 in 0ms cpu (0ms real) (~ rews/sec)
M:Marking --> null

No more solutions.
states: 6 rewrites: 5 in 0ms cpu (1ms real) (~ rews/sec)
```

The answer is yes, and the desired state is numbered 4. To see the sequence of rewrites that allowed us to reach this state we can type

```
Maude> show path 4 .
state 0, Marking: $ q q q
===[ rl $ => q a [label buy-a] . ]===>
state 2, Marking: q q q q a
===[ rl q q q q => $ [label change] . ]===>
state 3, Marking: $ a
===[ rl $ => c [label buy-c] . ]===>
state 4, Marking: a c
```

One can get just the sequence of labels of applied rules with a similar command:

```
Maude> show path labels 4 .
buy-a
change
buy-c
```

It is also possible to print out the current search graph generated by a search command using the command `show search graph`. After the above search we get

```
Maude> show search graph .
state 0, Marking: $ q q q
arc 0 ===> state 1 (rl $ => c [label buy-c] .)
arc 1 ===> state 2 (rl $ => q a [label buy-a] .)

state 1, Marking: q q q c

state 2, Marking: q q q q a
arc 0 ===> state 3 (rl q q q q => $ [label change] .)
```

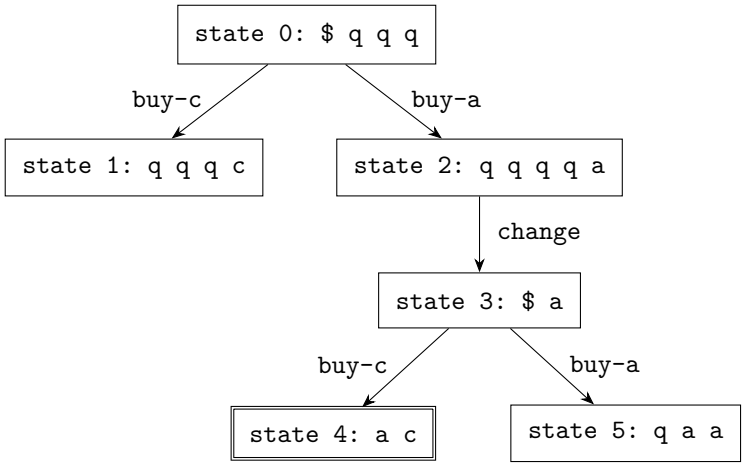


Figure 5.2: Graphical representation of search graph in example

```
state 3, Marking: $ a
arc 0 ===> state 4 (rl $ => c [label buy-c] .)
arc 1 ===> state 5 (rl $ => q a [label buy-a] .)

state 4, Marking: a c

state 5, Marking: q a a
```

This search graph is represented graphically in Figure 5.2.

From the same initial state, `$ q q q`, we can see if it is possible to reach a final state with an apple and more things, learning that there are exactly two possibilities:

```
Maude> search $ q q q =>! a M:Marking such that M:Marking =/= null .
search in SIMPLE-VENDING-MACHINE : $ q q q =>! a M:Marking
  such that M:Marking =/= null = true .

Solution 1 (state 4)
states: 6 rewrites: 6 in 0ms cpu (0ms real) (~ rews/sec)
M:Marking --> c

Solution 2 (state 5)
states: 6 rewrites: 7 in 0ms cpu (0ms real) (~ rews/sec)
M:Marking --> q a
No more solutions.
states: 6 rewrites: 7 in 0ms cpu (0ms real) (~ rews/sec)
```

In the following example with a different initial state, namely, `$ q q q q`, we are looking for intermediate states from which it is possible to get later either two apples (and two quarters left) or two cakes, getting exactly one solution.

```
Maude> search $ q q q q =>+ M:Marking
         such that M:Marking => a a q q /\ M:Marking => c c .
search in SIMPLE-VENDING-MACHINE : $ q q q q =>+ M:Marking
   such that M:Marking => q q a a /\ M:Marking => c c .
```

```
Solution 1 (state 1)
states: 2 rewrites: 10 in 0ms cpu (0ms real) (96153 rewrites/second)
M:Marking --> $ $

No more solutions.
states: 9 rewrites: 38 in 0ms cpu (0ms real) (95477 rewrites/second)
```

Sometimes it is necessary to impose a limit on the number of solutions searched for, since in general the number of such solutions could be infinite. In the previous examples there were only one or two solutions, so imposing a bound would not make any difference. But, returning to the coin generating (and thus nonterminating) vending machine module `VENDING-MACHINE`, the search space becomes now infinite, so it is important to be able to limit either the number of solutions sought or the depth of the search, or both.

We can look for different ways to use a dollar and three quarters to buy an apple and two cakes. First we ask for one solution, and then use the bounded `continue` command to see another solution. Note that here we use the search mode `=>+`, which means searching for states reachable by at least one rewrite. Searching for terminal states in the `VENDING-MACHINE` module is futile!

```
Maude> search [1] in VENDING-MACHINE : $ q q q =>+ a c c M:Marking .
search in VENDING-MACHINE : $ q q q =>+ a c c M .

Solution 1 (state 108)
states: 109 rewrites: 1857 in 0ms cpu (41ms real)(~rews/sec)
M --> q q q q

Maude> cont 1 .
Solution 2 (state 113)
states: 114 rewrites: 185 in 0ms cpu (4ms real) (~ rews/sec)
M --> null
```

We can also use the maximum depth optional argument, but if we put a too small depth, we do not get any solution:

```
Maude> search [, 4] $ q q q =>+ a c c M:Marking .
search [, 4] in VENDING-MACHINE : $ q q q =>+ a c c M .

No solution.
states: 66 rewrites: 875 in 10ms cpu (3ms real) (87500 rews/sec)
```

By increasing the depth to `10` we will get 98 solutions. If we are interested in only a few of those, we can set both bounds, like in the following example:

```
Maude> search [4, 10] $ q q q =>+ a c c M:Marking .
search [4, 10] in VENDING-MACHINE : $ q q q =>+ a c c M .

Solution 1 (state 108)
states: 109 rewrites: 1857 in 0ms cpu (7ms real) (~ rews/sec)
M --> q q q q

Solution 2 (state 113)
states: 114 rewrites: 2042 in 0ms cpu (7ms real) (~ rews/sec)
M --> null

Solution 3 (state 160)
states: 161 rewrites: 3328 in 0ms cpu (11ms real) (~ rews/sec)
```

```
M --> q q q q q

Solution 4 (state 164)
states: 165 rewrites: 3524 in 0ms cpu (12ms real) (~ rews/sec)
M --> q
```

If we insist now in the marking `M` being different from `null`, then one of the previous solutions is discarded, but we still get four solutions:

```
Maude> search [4, 10] $ q q q =>+ a c c M:Marking
        such that M:Marking =/= null .
search [4, 10] in VENDING-MACHINE : $ q q q =>+ a c c M
  such that M =/= null = true .

Solution 1 (state 108)
states: 109 rewrites: 1858 in 0ms cpu (5ms real) (~ rews/sec)
M --> q q q q

Solution 2 (state 160)
states: 161 rewrites: 3331 in 10ms cpu (13ms real) (333100 rews/sec)
M --> q q q q q

Solution 3 (state 164)
states: 165 rewrites: 3528 in 10ms cpu (14ms real) (352800 rews/sec)
M --> q

Solution 4 (state 175)
states: 176 rewrites: 3904 in 10ms cpu (15ms real) (390400 rews/sec)
M --> $ q q q q
```

In Chapter 11 we will see how the `search` command can be used to model check invariant properties of a concurrent system specified in Maude as a system module.

In case you forget to set a bound on the `search` command or on its continuation, you can also interrupt a search in progress by typing control-C. In this case one of two things will happen, depending on what Maude is doing at the instant you hit control-C. If Maude is not doing a rewrite, the command will exit. If Maude is doing a rewrite, you will end up in the debugger. In this latter case it is probably best to type `abort`, since the debugger has no special support for search at the moment. See Sections 20.1.3 and 23.14 for more information on the debugger.

The full syntax and different options for the `search` command and for all the other commands illustrated in this section are explained in detail in Chapter 23.

Chapter 6

Module Operations

Specifications and code should be structured in *modules* of relatively small size to facilitate understandability of large systems, increase reusability of components, and localize the effects of system changes. In Maude, these goals are achieved by means of a *module algebra* that supports parameterized programming techniques in the OBJ3 style [79] as well as the definition of *module hierarchies*, i.e., acyclic graphs of module importations; that is, each functional or system module can import other Maude modules as submodules. Since the submodule relation is *transitive*, we can in this way develop module *hierarchies*. Mathematically, we can think of such hierarchies as partial orders of *theory inclusions*, that is, the theory of the importing module contains the theories of its submodules as subtheories.

As in Clear [21], OBJ [79], and other specification languages in that tradition, the abstract syntax for writing specifications in Maude can be seen as given by *module expressions*, where the notion of module expression is understood as an expression that defines a new module out of previously defined modules by combining and/or modifying them according to a specific set of operations. In fact, structuring is essential in all specification languages, not only to facilitate the construction of specifications from already existing ones—with more or less flexible reusability mechanisms—but also for managing the complexity of understanding and analyzing large specifications. Maude supports module operations for summation, renaming, and instantiation of parameterized modules.

Section 6.1 introduces module importations and the different modes in which such importations can take place. Section 6.2 discusses the summation and renaming module expressions. Section 6.3 introduces parameterized programming, including the use of theories and views, the parameterization of functional and system modules, and the instantiation of parameterized modules. We refer to [44, 52, 53] for a deeper discussion on the semantics of the Maude module operations.

6.1 Module importation

Recall that a functional module M specifies a membership equational theory of the form $(\Sigma, E \cup A)$, with Σ its signature, A the equational attributes specified for its operators, and E its set of equations and memberships. A *submodule* M' of M is either a module directly imported by M, or a submodule of one of the modules directly imported by M. Then M' specifies a membership equational subtheory $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$. Specifically, we have three inclusions: $\Sigma' \subseteq \Sigma$, $E' \subseteq E$, and $A' \subseteq A$. Furthermore, since in Maude subsort-overloaded operators must have the *same* equational attributes, Maude will enforce that the inclusion $A' \subseteq A$ satisfies this

property.

In a similar way, a system module Q specifies a rewrite theory $(\Sigma, E \cup A, \phi, R)$. A submodule Q' of Q will likewise specify a rewrite subtheory $(\Sigma', E' \cup A', \phi', R') \subseteq (\Sigma, E \cup A, \phi, R)$. This means that we have inclusions $\Sigma' \subseteq \Sigma$, $E' \subseteq E$, $A' \subseteq A$ (again, with the same equational attributes for subsort-overloaded operators), $\phi' \subseteq \phi$, and $R' \subseteq R$, where $\phi' \subseteq \phi$ is an inclusion of functions and means that the freezing function ϕ *extends* the function ϕ'. Note that Q' could be a functional module, which is then understood as the rewrite theory $(\Sigma', E' \cup A', \phi', \emptyset)$, where ϕ' specifies whatever freezing information has been given to the operators of Σ' in Q'. A system module cannot be imported into a functional module.

In Maude, a module—any module expression giving rise to a module—can be imported as a submodule of another in three different modes: `protecting`, `extending`, or `including`. This is done with the syntax declarations

`protecting` ⟨*ModuleExpression*⟩ .
`extending` ⟨*ModuleExpression*⟩ .
`including` ⟨*ModuleExpression*⟩ .

which can be abbreviated, respectively, to

`pr` ⟨*ModuleExpression*⟩ .
`ex` ⟨*ModuleExpression*⟩ .
`inc` ⟨*ModuleExpression*⟩ .

In addition to being allowed as arguments of a `protecting`, `extending`, or `including` importation, module expressions can also appear as the source or target of a view (see Section 6.3.2), or as the parameter of a module, provided the top level is a theory (see Section 6.3.3).

Each of the importation modes places specific semantic constraints on the corresponding inclusion between the theory of the submodule and that of the supermodule. The user must be aware that, as explained later, *the Maude system does not check that these constraints are satisfied*, that is, the different modes of importation can be understood as promises by the user, which would need to be proved by him/herself. Although those importation modes have no effect operationally, they do crucially affect the interpretation given to a module by the theorem proving tools. If a user is doubtful about the appropriate importation mode the default should be to use the `including` mode, which places weaker requirements on the importation.

Importation statements take a module expression as argument, which may be a module name, the summation of module expressions, the renaming of a module expression, or the instantiation of a parameterized module expression. Modules are constructed for each subexpression of a module expression, and so each submodule signature must be legal. Modules and module expressions are cached both to save time and so that the same module corresponding to a module expression will not be imported twice via a diamond of imports. Mutually or self recursive imports occurring through module expressions are detected and disallowed. Cached modules generated by module expressions that no longer have any users (if the module(s) containing the module expression have been replaced) are deleted. When a module M used in a module expression is modified, any modules generated for module expressions that depend on M are deleted and any modules that depend on M are reevaluated if you attempt to use them. Here the notion of "depends on" is transitive through arbitrary nesting of importation and module expressions.

In addition to being imported by the explicit importation statements we have just introduced, modules can be imported in an implicit way (also in the three possible modes) by means of commands `set protect/extend/include` *`module`* `on/off`; see more details in Section 23.15 and the detailed example in Section 7.1.

6.1.1 Protecting

Importing a module M' into M in `protecting` mode intuitively means that *no junk and no confusion* are added to M' when we include it in M. For example, we may import the module `NAT` of natural numbers into a module `FOO`. "Junk" would be added to `NAT` if in `FOO` we have new ground terms in canonical form in any of the sorts in `NAT`, namely `Nat` and `NzNat`. For example, `FOO` may have declared a constant `infinity` of sort `NzNat` to which no equations apply. "Confusion" would be added if different natural numbers are now identified. For example, if `FOO` contains the equation `s s 0 = 0`, then all even numbers will be identified with `0` and all odd numbers with `s 0`.

Let us explain the semantics of the `protecting` relation in more detail for functional modules M' and M, where M' has been imported as a submodule in `protecting` mode, either by an explicit protecting importation in M, or transitively through one of M's submodules. Let $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ be the theory inclusion defined by the module inclusion $M' \subseteq M$. Notice that the existence of the inclusions $\Sigma' \subseteq \Sigma$, $E' \subseteq E$, and $A' \subseteq A$ means that for each sort s' in Σ' there is a well-defined function

$$q_{s'} : T_{\Sigma'/E' \cup A', s'} \longrightarrow T_{\Sigma/E \cup A, s'}$$

mapping the equivalence class $[t]_{E' \cup A'}$ of a ground term t to the equivalence class $[t]_{E \cup A}$. By definition, the submodule inclusion $M' \subseteq M$ is *protecting* if and only if for each sort s' in Σ' the above function is *bijective*. This captures mathematically the "no junk" (surjectivity) and "no confusion" (injectivity) ideas. Under our ground Church-Rosser and termination assumptions for M' and M this also means that the canonical form of any ground Σ'-term t in M' *that has a sort* in Σ' must be the same as its canonical form in M. The requirement that t must have a sort is crucial. We do not require that for k' a kind the map

$$q_{k'} : T_{\Sigma'/E' \cup A', k'} \longrightarrow T_{\Sigma/E \cup A, k'}$$

is bijective. The reason is that the notion of *defined function*—that is, an operator that disappears and leaves just a term with constructors—is only meaningful when the result has a sort. The same operator may not disappear for error terms at the kind level. That is, in the module M extending M' there may easily be *new error terms* of kind k' created by new operators in M. For example, if we import the module `NAT` into the module `RAT` of rational numbers, the sorts `Nat` and `Rat` belong to the same kind, but there are now new error terms in the kind, such as `3 + 7/0`. Therefore, we should not care about "error junk" being added by a supermodule at the kind level, provided that the sorts themselves are protected.

For system modules the `protecting` requirement is interpreted exactly as before as far as their underlying equational theories are concerned. That is, if Q protects Q' and the associated theory inclusion is $(\Sigma', E' \cup A', \phi', R') \subseteq (\Sigma, E \cup A, \phi, R)$, then the equational theory inclusion $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ must be *protecting*. We furthermore require that for any two ground Σ'-terms t and t' we can reach t' from t by a sequence of rewrites in the module M' *if and only if* we can do so in the module M; that is, for ground terms in M' we require that the *reachability relation* is not altered by the supermodule.

Of course, the `protecting` assertion cannot be checked by Maude at runtime: it requires inductive theorem proving. Using the proof techniques in [17] together with an inductive theorem prover for membership equational logic and a Church-Rosser checker such as those described in [54, 56, 30] (which are available in the Maude formal tool environment together with other useful tools for termination and sufficient completeness), this can be done for functional modules. Using the fact that initial models of rewrite theories are also models of equational theories [19], similar proof techniques could be developed to prove the protecting relation between rewrite theories.

6.1.2 Extending

A weaker, yet substantial, requirement about a module importation is expressed by the keyword `extending`. Intuitively, the idea is to allow "junk," but to rule out confusion. Extending importations may appear naturally in situations in which the data of some sort is extended with new data elements, yet not identifying previously defined data, like adding a new constant `infinity` to the natural numbers in a module importing `NAT`. As another example, when defining the semantics of a programming language in Maude, we can have from the beginning a sort `Program`, and define incrementally the syntax of programs in several modules, say, `EXPRESSION`, `STATEMENT`, `PROCEDURE`, and so on. This will typically give rise to a family of `extending` module importations as more syntax is added.

For functional modules M' and M, where M' has been imported as a submodule in `extending` mode, either by an explicit extending importation in M, or transitively through one of M's submodules, if $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ is the theory inclusion defined by the module inclusion $M' \subseteq M$, the `extending` requirement means that for each sort s' in Σ' the function

$$q_{s'} : T_{\Sigma'/E' \cup A', s'} \longrightarrow T_{\Sigma/E \cup A, s'}$$

is *injective*. For system modules the `extending` requirement is interpreted exactly as before as far as their underlying equational theories are concerned. That is, if Q extends Q' and the associated theory inclusion is $(\Sigma', E' \cup A', \phi', R') \subseteq (\Sigma, E \cup A, \phi, R)$, then the equational theory inclusion $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ must be *extending*. We furthermore require that for any two ground Σ'-terms t and t' we can reach t' from t by a sequence of rewrites in the module M' *if and only if* we can do so in the module M; that is, for ground terms in M' the *reachability relation* is not altered by the supermodule.

Under the Church-Rosser and termination assumptions, the `extending` $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ requirement is a form of *conservative extension* requirement, in the sense that it implies that for any Σ' ground terms t and t' that have a sort in $(\Sigma', E' \cup A')$, $E' \cup A'$ proves $t = t'$ if and only if $E \cup A$ proves $t = t'$. In addition, for system modules it further implies that for any two ground Σ'-terms t and t' the reachability relation is not altered by the extension. In summary, equality and reachability are conservatively preserved for ground terms.

Note that the `extending` relation does *not* destroy protecting importations further down the hierarchy. That is, if M imports M' in `extending` mode, but M' imports M'' in `protecting` mode, then M still imports M'' in `protecting` mode, *not* in `extending` mode. If we do not want M to protect M'' (because this is indeed violated), then we have to say so by explicitly giving an `extending` importation declaration for M'' in M.

6.1.3 Including

The most general form of module importation is provided by the `including` keyword. No requirements are made in an `including` importation about maps of the form $q_{s'}$: there can now be junk (lack of surjectivity) and/or confusion (lack of injectivity). Likewise, for system modules it is not anymore required that the reachability relation between ground terms in the submodule is preserved. The `including` keyword does however impose *some* requirements. First of all, there is the requirement that the equational attributes of subsort-overloaded operators must be the same. Furthermore, the `including` relation does *not* destroy protecting or extending importations further down the hierarchy. That is, if M imports M' in `including` mode, but M' imports M'' in `protecting` (resp. `extending`) mode, then M still imports M'' in `protecting` (resp. `extending`) mode, *not* in `including` mode. If we do not want M to protect or extend M'' (because this is indeed violated), then we have to say so by explicitly giving an `including` importation declaration for M'' in M.

Given that, as already mentioned, there is no difference at runtime between the different modes of importation because the Maude system does not check the corresponding requirements, from a pragmatic point of view, when a user is doubtful about the appropriate importation mode, the best idea is to use the `including` mode so that at least no false assertion is made.

6.1.4 Default conventions in module importations

We have already explained our default convention when a submodule M_0 is imported indirectly and transitively into M through the direct importation by M of a module M_1 that itself imports M_0. Then, whatever was the mode (`protecting`, `extending`, or `including`) in which M_0 was imported by M_1 is also, by default, the mode in which M_0 is imported by M, *unless* M contains an explicit declaration importing M_0 in a different mode. We now explain what our default convention is in the case of diamond importations.

We talk of a *diamond importation* of M_0 by M, when M_0 is imported indirectly by M through the direct importation of two or more different modules, say $M_1, M_2, \ldots, M_k$. The problem now is that M_0 can be imported by each of the modules $M_1, M_2, \ldots, M_k$ in *different* modes. For example, M_1 could import it in `protecting` mode, M_2 in `extending` mode, M_3 in `including` mode, and so on. What should now be the default convention for the mode in which M imports M_0? We adopt a convention that is consistent with a *logical* understanding of such importation declarations. Indeed, such declarations impose semantic constraints of decreasing strength; that is, we have:

$$\texttt{protecting}\ M_0 \ \Rightarrow\ \texttt{extending}\ M_0 \ \Rightarrow\ \texttt{including}\ M_0.$$

The default convention consistent with this logical reading is that *the strongest mode wins*, i.e., `protecting` prevails over `extending`, which itself prevails over `including`. This is because we view the set of all such importing mode declarations as a *conjunction*, and exploit the logical equivalence between $A \Rightarrow B$ and $(A \wedge B) \Leftrightarrow A$.

Note that this "strongest wins" default mode may not always be the correct or intended mode in which M *should* import M_0. Sometimes it may not be, and then the user should *overrule* the default convention by declaring explicitly a different mode in which M imports M_0. A pragmatic reason why this need for overruling the default mode may arise is that, although a weaker mode of importation (say `extending`) does not logically preclude such an importation also satisfying a stronger one (say `protecting`), in practice, when we declare an importation in a weaker mode it can often be because we know or suspect that it fails to satisfy a stronger mode. For example, when we say "`extending`" we may often mean "`extending` and *not* `protecting`."

6.1.5 Some module hierarchy examples

Prime numbers sieve

Section 4.4.7 included a functional module specifying the sieve of Eratosthenes to calculate prime numbers.

```
fmod SIEVE is
  protecting NAT .
  sort NatList .
  subsort Nat < NatList .
  ...
endfm
```

The predefined module of natural numbers (see Section 7.2) is imported in `protecting` mode. This is justified because the elements of sort `Nat` are used to generate the lists of natural numbers by means of a subsort declaration and also as arguments of other operators. However, no new operator of result sort `Nat` is added in the `SIEVE` module, and all the equations in this module identify elements of sort `NatList` without identifying different natural numbers.

Vending machine

The vending machine example in Section 5.1 was presented in a modular way, by separating the underlying signature defining the states of the machine from the rules defining the corresponding transitions.

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  rl [add-q] : M => M q .
  ...
endm
```

It is important to notice that in this example the importation mode cannot be either `protecting` or `extending`, because those modes require preservation of the reachability relation, which clearly is not the case when adding (non-identity) rewrite rules to a functional module (where the reachability relation is the identity).

Bank accounts and object configurations

Later, in Section 8.1, devoted to the definition of configurations of objects and messages for object-based programming, we will present several modules where additional data are introduced in order to run some tests. For example, the following module introduces three new constants to be used as object identifiers, and a new constant to be used as a test configuration. This configuration constant is identified with a term of sort `Configuration` in the imported module `BANK-ACCOUNT` by means of an equation whose righthand side is omitted below. However, constants `A-001`, `A-002`, and `A-003` are new data elements, i.e., junk, of sort `Oid`. The sort `Oid` was declared in the module `CONFIGURATION`, but since it was imported in `including` mode in `BANK-ACCOUNT`, it is not necessary to import it in a different mode. Therefore, the appropriate importation mode is `extending`.

```
mod BANK-ACCOUNT-TEST is
  ex BANK-ACCOUNT .
  ops A-001 A-002 A-003 : -> Oid .
  op bankConf : -> Configuration .
  eq bankConf = ... .
endm
```

The following example, from Section 8.3, is more interesting, in that it introduces new sorts `MsgBody` and `Request`, not just new constants for a sort in the imported module. Still, the appropriate importation mode is `extending` because there are no new rewrite rules and no equations, and thus no confusion between elements in imported sorts is introduced.

```
mod DATA-AGENTS-CONF is
  ex CONFIGURATION .
  sort MsgBody .
  op msg : Oid Oid MsgBody -> Msg [ctor message] .
  sort Request .
  op w4 : Oid Oid MsgBody -> Request [ctor] .
endm
```

There are several other modules in Chapter 8 illustrating the use of the `extending` mode in importing modules, like `BANK-MANAGER-TEST`, `TICKER-TEST`, `TICKER-FACTORY-TEST`, and `AGENT-TEST`; see Figures 8.1, 8.2, and 8.3.

Hierarchy of predefined modules

A more complex acyclic importation graph corresponds to the hierarchy of predefined modules for basic data types, described later in Chapter 7 and shown in Figure 7.1, where all the importations are in `protecting` mode.

6.2 Module summation and renaming

6.2.1 The summation module expression

The summation module operation creates a new module that includes all the information in its summands. The syntax for a summation of module expressions is

ModuleExpression `+` *ModuleExpression*

with `+` associative and commutative.

Summation expressions are flattened before being evaluated, so that `A + (B + C)` and `(C + A) + B` both create a single new module `A + B + C`, The evaluation of a summation module expression results in the creation of a new module, with such a module expression as its name, which imports the module expressions being combined. The new module will be generated having one type or another, depending on the types of the arguments of the summation module expression. A summation is a functional module if all the summands are functional modules and a system module otherwise.

Although the use of the summation module expression is more interesting in combination with other module expressions, let us consider as an example the following module, in which the union of the predefined `FLOAT` and `STRING` modules (see Chapter 7) are imported together in `protecting` mode to illustrate its use.

```
fmod FLOAT-STRING is
  protecting FLOAT + STRING .
  ...
endfm
```

Notice that a declaration

```
protecting A + B .
```

is *not* equivalent to a sequence of declarations

```
protecting A .
protecting B .
```

because in general the modules `A` and `B` may share sorts and operators. The same happens with `extending` declarations, for the same reason. However, a declaration of the form

```
including A + B .
```

is indeed equivalent to a sequence of declarations

```
including A .
including B .
```

6.2.2 Module renaming

The syntax of a renaming module expression is

⟨*ModuleExpression*⟩ `*` `(` ⟨*Renaming*⟩ `)`

where ⟨***Renaming***⟩ is a comma-separated sequence of renaming items of the forms:

`sort` ⟨*identifier*⟩ `to` ⟨*identifier*⟩
`op` ⟨*identifier*⟩ `to` ⟨*identifier*⟩
`op` ⟨*identifier*⟩ `to` ⟨*identifier*⟩ `[` ⟨*attribute-set*⟩ `]`
`op` ⟨*identifier*⟩ `:` ⟨*type-list*⟩ `->` ⟨*type*⟩ `to` ⟨*identifier*⟩
`op` ⟨*identifier*⟩ `:` ⟨*type-list*⟩ `->` ⟨*type*⟩ `to` ⟨*identifier*⟩
`[` ⟨*attribute-set*⟩ `]`
`label` ⟨*identifier*⟩ `to` ⟨*identifier*⟩

Renaming (`_*(_)`) binds tighter than summation (`_+_`), and it groups to the left. Note that, in addition to the typical renamings of sorts and operators, renaming of labels is also supported (which may be useful for metalevel applications). Note also how the renaming of operators allows changing the attributes of the operator being renamed. The only attributes currently allowed in operator maps are `prec`, `gather`, and `format`. The idea is that when you rename an operator, the old syntactic properties may no longer be legal and are reset to defaults, unless you explicitly set them with these attributes; for example, when a change in the syntax of the operator could cause a parsing different from the intended one. Let us see an example in which modifying the grammatical attributes of an operator is useful. Consider the following module defining lists of natural numbers with a `max` operator returning the greatest of the elements in a list of natural numbers.

```
fmod NAT-LIST-MAX is
  pr NAT .
  sort NeNatList .
  subsort Nat < NeNatList .
  op __ : NeNatList NeNatList -> NeNatList [ctor assoc] .
  op max : NeNatList -> Nat .
  var N : Nat .
  var NL : NeNatList .
  eq max(N) = N .
  eq max(N NL) = if N > max(NL) then N else max(NL) fi .
endfm
```

We may obtain the maximum of a list of natural numbers as follows.

```
Maude> red max(4 2 5 3) .
result NzNat: 5
```

Suppose now that we want to change the syntax of the function `max` in the `NAT-LIST-MAX` module above to `maximum_`.

```
fmod NAIVE-NAT-LIST-MIXFIX-MAX is
  pr NAT-LIST-MAX * (op max : NeNatList -> Nat to maximum_) .
endfm
```

We can do the following reduction:

```
Maude> red maximum 2 3 4 1 .
result NeNatList: 2 3 4 1
```

This result may seem strange, but it makes perfect sense. What has happened is that it has been parsed as `(maximum 2) 3 4 1`, the only possible parse given the default precedence values

and gathering patterns assigned. Since by default `maximum_` has precedence 15 and gathering `E`, it cannot take the list `2 3 4 1` as argument because `__` has precedence 41. However, since `__` has gathering `e E`, `maximum 2` is a valid argument for it (see Section 3.9 for a detailed discussion on the use of precedence values and gathering patterns and their default values). We can of course obtain the intended result by placing parentheses around the set of numbers,

```
Maude> red maximum (2 3 4 1) .
result NzNat: 4
```

but it is more convenient to change the precedence values of the operators. We can, for example, raise the precedence of `maximum_`.

```
fmod NAT-LIST-MIXFIX-MAX is
  pr NAT-LIST-MAX
       * (op max : NeNatList -> Nat to maximum_ [prec 41]) .
endfm
```

having then the following reduction.

```
Maude> red maximum 2 3 4 1 .
result NzNat: 4
```

Notice that if `maximum_` has precedence `41`, then `(maximum 2) 3 4 1` is no longer a valid parse.

A renaming can be considered as a function that, given a module M and a list of mappings S, returns a copy of the module, with such a module expression as its name, in which the names of sorts, operators, etc. are changed as indicated by the mappings. However, renaming a module that has imports is a subtle issue. Given a structured specification, the renaming not only causes the creation of a copy of the top module in the structure, but renames also the part of the submodule structure that is affected by the renaming. For any other submodule M' in the structure which is affected by the mappings, a renamed copy of it is generated with name M' `*` `(`S'`)`, where S' is the subset of mappings in S that affect M'.

A module expression A `*` `(`R`)` evaluates to A if A has no content that is affected by the renaming R. Otherwise A `*` `(`R`)` evaluates to a new module A `*` `(`R'`)` where R' is obtained by deleting those renaming items that do not affect A, and canonizing the types in operator renamings with respect to A (see below). If A imports modules B and C, A `*` `(`R'`)` will import modules obtained by evaluating B `*` `(`R'`)` and C `*` `(`R'`)`.

There are some subtle cases. Consider for example the following three modules:

```
fmod RENAMING-EX-A is
  sort Foo .
  op a : -> Foo .
  op f : Foo -> Foo .
endfm

fmod RENAMING-EX-B is
  including RENAMING-EX-A .
  sort Bar .
  subsort Foo < Bar .
  op f : Bar -> Bar .
endfm

fmod RENAMING-EX-C is
  inc RENAMING-EX-B * (op f : Bar -> Bar to g) .
endfm
```

Here, the operator `f` in the module `RENAMING-EX-A` looks as though it is not affected by the renaming in the module `RENAMING-EX-C`, but because of the subsort declaration `Foo < Bar` in `RENAMING-EX-B`, it should be renamed for consistency. This is internally handled by the Maude system by canonizing the type `Bar` occurring in the renaming

```
op f : Bar -> Bar to g
```

to the kind expression `[Foo,Bar]`, which includes *all* of the sorts in the kind `[Bar]` occurring in `RENAMING-EX-B`. Thus, the module expression

```
RENAMING-EX-B * (op f : Bar -> Bar to g)
```

evaluates to a new module

```
RENAMING-EX-B * (op f : [Foo,Bar] -> [Foo,Bar] to g)
```

which includes the module expression

```
RENAMING-EX-A * (op f : [Foo,Bar] -> [Foo,Bar] to g)
```

which evaluates to a new module

```
RENAMING-EX-A * (op f : [Foo] -> [Foo] to g)
```

in which `f` has been renamed.

In general, `*` does not distribute over `+`. Consider this other example:

```
fmod RENAMING-EX-D is
  sorts Foo Bar .
endfm


fmod RENAMING-EX-E is
  inc RENAMING-EX-D .
  op f : Foo -> Foo .
endfm


fmod RENAMING-EX-F is
  inc RENAMING-EX-D .
  subsort Foo < Bar .
  op f : Bar -> Bar .
endfm
```

It is *not* the case that the module expressions

```
(RENAMING-EX-E + RENAMING-EX-F) * (op f : Bar -> Bar to g)
```

and

```
(RENAMING-EX-E * (op f : Bar -> Bar to g))
  + (RENAMING-EX-F * (op f : Bar -> Bar to g))
```

evaluate to the same module, because in the latter the operator `f` occurring in `RENAMING-EX-E` will not be renamed, since `f : Bar -> Bar` does not occur in `RENAMING-EX-E`.

Operators with the `poly` attribute are only affected by operator renamings that do not specify types. Renaming a module does not change its module type, that is, renamed functional modules (resp. system modules) remain functional (resp. system).

6.3 Parameterized programming

Theories, parameterized modules, and views are the basic building blocks of parameterized programming [21, 79]. As in OBJ, a *theory* defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter.

A *parameterized module* is a module with one or more *parameters*, each of which is expressed by means of one theory, that is, modules can be parameterized by one or more theories. If we want, e.g., to define a list or a set of elements, we may define a module `LIST` or `SET` parameterized by a theory expressing the requirements on the type of the elements to store in such data structures. Thus, theories are used to declare the interface requirements for parameterized modules. In the case of lists and sets we do not need any requirement on the data elements, and therefore we may use the trivial theory `TRIV`, with just a sort `Elt`, as parameter of such modules; but in other cases, say search trees or sorted lists, we may require, e.g., a particular operator, an order relation, or an equivalence relation, in which cases we shall need to use the appropriate theories describing the specific requirements.

The instantiation of the formal parameters of a parameterized module with actual parameter modules or theories requires a *view* mapping entities from the formal interface theory to the corresponding entities in the actual parameter module. Views can also be parameterized, which, as we can see in Section 6.3.7, may greatly improve reusability of specifications

6.3.1 Theories

Theories are used to declare module interfaces, namely the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation. As for modules, Maude supports two different types of theories: *functional theories* and *system theories*, with the same structure of their module counterparts, but with a different semantics. Functional theories are declared with the keywords `fth ... endfth`, and system theories with the keywords `th ... endth`. Both of them can have sorts, subsort relationships, operators, variables, membership axioms, and equations, and can import other theories or modules. System theories can also have rules. Although there is no restriction on the operator attributes that can be used in a theory, there are some subtle restrictions and issues regarding the mapping of such operators (see Section 6.3.2).

Like functional modules, *functional theories* are membership equational logic theories, but they do not need to be Church-Rosser and terminating, and therefore some or all of their statements may be declared with the `nonexec` attribute. Theories have a *loose* semantics, in the sense that any algebra satisfying the equations and membership axioms in the theory is an acceptable model. However, functional theories *can be executed in exactly the same way as functional modules*; that is, the equations and membership axioms not having the `nonexec` attribute should be Church-Rosser and terminating, and can be executed by equational simplification, whereas the statements declared as `nonexec` can be arbitrary and can only be executed in a controlled way at the metalevel. System theories have a similar loose interpretation, but are treated just as system modules for executability purposes. Theories are then allowed to contain rules and equations which, if declared with the `nonexec` attribute, can be arbitrary, that is, can have variables in their righthand sides or conditions that may not appear in their corresponding lefthand sides and do not obey the admissibility conditions in Sections 4.6 and 5.3. Similarly, conditional membership axioms may have variables in their conditions that do not appear in their head membership assertions. Also, the lefthand side may be a single variable.

Let us begin by introducing the functional theory `TRIV`, which requires just a sort.

```
fth TRIV is
```

```
  sort Elt .
endfth
```

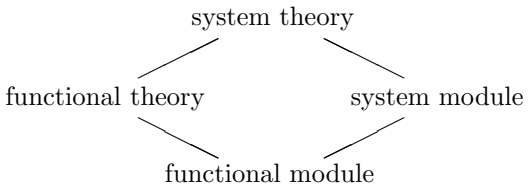
The theory TRIV is used very often, for instance in the definition of data structures, such as lists, sets, trees, etc., of elements of some type with no specific requirement; in these cases, it is common to define a module, say LIST, SET, TREE, etc., parameterized by the TRIV theory (see Section 6.3.3). The theory TRIV is predefined in Maude, together with several useful views from TRIV to other predefined modules and theories (see Section 7.12.1).

But we can define more interesting theories. For example, the theory of monoids, with an associative binary operator with identity element 1, can be specified as follows:

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

Notice the importation of the theory TRIV into the MONOID theory. As for modules, it is possible to structure our theories by importing other theories and modules (and in general module expressions involving theories and modules) into theories. However, a theory cannot be imported into a module: theories can only be used as *parameters* of modules. Also, theories do not have automatic importation as modules do (e.g., BOOL, as described in Section 7.1).

Modules and theories can be combined in module expressions (they can be summed, for example), and modules can be imported into theories. Basically, we have a lattice



where summation corresponds to join, and where a module or theory may only import a submodule or subtheory of lesser or equal type.

Although the importation of a module into a theory can be done in any mode, a theory can only be imported in `including` mode into another theory. The `including` importation of a theory into another theory keeps its loose semantics. However, the importation of a theory into another one in `protecting` or `extending` mode would imply additional semantic requirements; such modes of importation are ruled out.[1] On the other hand, although a module keeps its initial interpretation when imported into a theory in `protecting` or `extending` modes, it looses it if imported in `including` mode.

Let us see a few examples illustrating all this.

The theory of commutative monoids can be defined just as the theory of monoids; the binary operator _+_ is now declared associative, commutative, and has 0 as its identity element.

```
fth +MONOID is
  including TRIV .
  op 0 : -> Elt .
  op _+_ : Elt Elt -> Elt [assoc comm id: 0] .
endfth
```

[1] If a theory is imported using a mode other than `including`, the system gives an error message saying that the mode is being treated as if it were `including`. Other illegal importations give an error message saying that they are being ignored.

The theory of semirings can be expressed as follows.

```
fth SEMIRING is
  including MONOID .
  including +MONOID .
  vars X Y Z : Elt .
  eq X (Y + Z) = (X Y) + (X Z) [nonexec] .
  eq (X + Y) Z = (X Z) + (Y Z) [nonexec] .
endfth
```

Note the use of the `nonexec` attribute, and note also that given the semantics of theory inclusions, there is no difference between having a structured theory or one flat theory including all the declarations.[2] For example, the theory of commutative rings can be defined directly as follows:

```
fth RING is
  sort Ring .
  ops z e : -> Ring .
  op _+_ : Ring Ring -> Ring [assoc comm id: z] .
  op _*_ : Ring Ring -> Ring [assoc comm id: e] .
  op -_ : Ring -> Ring .
  vars A B C : Ring .
  eq A + (- A) = z [nonexec] .
  eq A * (B + C) = (A * B) + (A * C) [nonexec] .
endfth
```

but could also be defined as a structured theory including the theories of commutative groups and commutative monoids (renamed if necessary), to which the distributivity axiom is added.

As mentioned above, the `including` importation of a theory into another theory keeps its loose semantics. However, if the imported theory contains a module, which therefore must be interpreted with an initial semantics (see Section 5.3), then that initial semantics is maintained by the importation. For example, in the definition of the `TAOSET` theory below, the declaration `protecting BOOL` ensures that the initial semantics of the functional module for the Booleans is preserved, which is in fact a crucial requirement.

Let us consider now a hierarchy of theories for partially and totally ordered sets. The most basic theory specifies a transitive and antisymmetric order `_<_` on a set:

```
fth TAOSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
  ceq X = Y if X < Y /\ Y < X [nonexec label antisymmetric] .
endfth
```

By adding irreflexivity to `TAOSET` we get a theory specifying a strict partial order:

```
fth SPOSET is
  including TAOSET .
  var X : Elt .
  eq X < X = false [nonexec label irreflexive] .
```

[2] The only exception to this semantic equivalence between structured theories and their flattened form is the case in which a theory imports some modules, since any of the `protecting` or `extending` initiality requirements of the imported module and its submodules *must be preserved*. Those requirements would be lost if the whole structure were to be flattened.

```
endfth
```

Notice that in this case antisymmetry is implied by irreflexivity and transitivity. Of course, there are different ways of presenting a theory, and in particular one can always write the corresponding flat theory with only the axioms for irreflexivity and transitivity. In the presentation above, the initial semantics of `BOOL` when it is imported in `protecting` mode in `TAOSET` is preserved when the latter is included in `SPOSET`. The same will hold in further importations in this hierarchy of order theories.

On the other hand, by adding reflexivity to `TAOSET` we get a theory specifying a non-strict partial order. Notice the renaming in the importation, so that the name of the order `_<=_` reflects its reflexivity.

```
fth NSPOSET is
  including TAOSET * (op _<_ to _<=_) .
  var X : Elt .
  eq X <= X = true [nonexec label reflexive] .
endfth
```

Having both `_<_` and `_<=_` available together is useful in some applications. There are standard ways of associating a strict partial order with a non-strict partial order and the other way around:

- from $a < b$, one can define $a \leq b$ as equivalent to $a < b$ or $a = b$; and
- from $a \leq b$, one can define $a < b$ as equivalent to $a \leq b$ and $a \neq b$.

These equivalences can be expressed as Maude theories as follows, where we use the same name for both theories because they are equivalent, that is, we have two different presentations of the same theory and in what follows we will not care about which version of `POSET` is used.

```
fth POSET is
  including SPOSET .
  op _<=_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X <= X = true [nonexec] .
  ceq X <= Y = true if X < Y [nonexec] .
  ceq X = Y if X <= Y /\ X < Y = false [nonexec] .
endfth

fth POSET is
  including NSPOSET .
  op _<_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X < X = false [nonexec] .
  ceq X <= Y = true if X < Y [nonexec] .
  ceq X = Y if X <= Y /\ X < Y = false [nonexec] .
endfth
```

Notice that the axioms are almost the same in both presentations of `POSET`, but, while the first presentation defines the reflexive order `_<=_` in terms of the irreflexive one `_<_`, the second presentation defines the irreflexive order `_<_` in terms of the reflexive one `_<=_`.

To each of the previous theories we can add an axiom requiring the order to be total (or linear), that is, two different elements have to be related one way or the other. In this way, we have the following theories for a strict total order, a non-strict total order, and a total order with both operations.

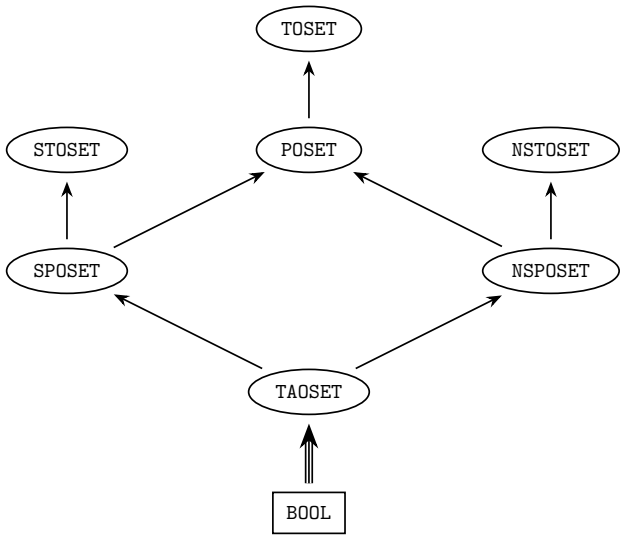


Figure 6.1: Hierarchy of order theories

```
fth STOSET is
  including SPOSET .
  vars X Y : Elt .
  ceq X = Y if X < Y = false /\ Y < X = false [nonexec label total] .
endfth

fth NSTOSET is
  including NSPOSET .
  vars X Y : Elt .
  ceq X <= Y = true if Y <= X = false [nonexec label total] .
endfth

fth TOSET is
  including POSET .
  vars X Y : Elt .
  ceq X <= Y = true if Y <= X = false [nonexec label total] .
endfth
```

As already mentioned above, the requirement ensuring the initial semantics of `BOOL` when it is protected in `TAOSET` is then preserved by the remaining theories when `TAOSET` is included in them via a chain of `including` importations. In fact, we are dealing with structures in which part of them, not only the top theory, has a loose semantics, while other parts contain modules with an initial semantics.

This hierarchy of order theories is displayed in Figure 6.1, where we represent by boxes the modules (with initiality constraints), by ovals the theories (with loose semantics), by triple arrows the `protecting` importations, and by single arrows the `including` importations.

Finally, as an example of a system theory, let us consider the theory `CHOICE` of bags of elements with a choice operator defined on the bags by a rewrite rule that nondeterministically picks up one of the elements in the bag. We can specify this theory as follows, where we have a sort `Bag` declared as a supersort of the sort `Elt`.

```
th CHOICE is
  sorts Bag Elt .
  subsort Elt < Bag .
  op empty : -> Bag .
  op __ : Bag Bag -> Bag [assoc comm id: empty] .
  op choice : Bag -> Elt .
  var E : Elt .
  var B : Bag .
  rl [choice] : choice(E B) => E .
endth
```

6.3.2 Views

We use *views* to specify how a particular target module or theory is claimed to satisfy a source theory. In general, there may be several ways in which such requirements might be satisfied, if at all, by the target module or theory; that is, there can be many different views, each specifying a particular *interpretation* of the source theory in the target. Each view declaration has an associated set of *proof obligations*, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds true in the target. Since the target can be a module interpreted initially, verifying such proof obligations may in general require inductive proof techniques of the style supported for Maude's logic in [30], and for which tools in the Maude formal environment can be used. *Such proof obligations are not discharged or checked by the system.*

In the definition of a view we have to indicate its name (which has to be a single identifier, as defined in Section 3.1), the source theory, the target module or theory, and the mapping of each sort and operator in the source theory. The name space of views is separate from the name space of modules and theories, which means that, e.g., a view and a module could have the same name. In fact, we shall see below how we recommend naming inclusion views as the target theory. The source and target of a view can be any module expression, with the source module expression evaluating to a theory and the target module expression evaluating to a module or a theory.

The syntax for views is as follows:

```
view ⟨ViewName⟩ from ⟨Source⟩ to ⟨Target⟩ is
  ⟨Mappings⟩
endv
```

The mapping of a sort in the source theory to a sort in the target module or theory is expressed with syntax

```
sort ⟨identifier⟩ to ⟨identifier⟩ .
```

For each sort S in the source theory, there must exist a sort S' in the target module or theory which is its mapping under the view; unmentioned sorts get the identity mapping. Furthermore, if sorts S and T in the source theory are in the same kind, then their mappings S' and T' under the view must be in the same kind in the target module or theory. Finally, if S is a subsort of T, then it must be true that S' is a subsort of T'.

The mapping of operators is expressed with syntax

```
op ⟨identifier⟩ to ⟨identifier⟩ .
op ⟨identifier⟩ : ⟨type-list⟩ -> ⟨type⟩ to ⟨identifier⟩ .
op ⟨op-expr⟩ to term ⟨term⟩ .
```

In the first case, where only an operator identifier is given, the map affects all operators with the same name. Existence of appropriate operators in the target is checked for. In the second case, when explicit arity and coarity are given, the operator map affects not only the operators with such arity and coarity, but also the entire family of subsort-overloaded operators (see Section 3.6) associated with the given operator. The third case is similar to the second one, but instead of mapping the operator to another operator, it is mapped to a given term with variables; ⟨ *op-expr* ⟩ is a term consisting of a single operator applied to variables—declared either on-the-fly or with variable declarations in the same view—and the target term is any term with variables, those in the source ⟨ *op-expr* ⟩ in the corresponding sorts resulting from the mapping. See below for more details and examples.

Maps must preserve the arities and the types of operators, and sort maps and operator maps must be compatible. For each operator f `:` $S_1 \dots S_n$ `->` T in the source theory there must exist an operator f' `:` $S'_1 \dots S'_n$ `->` T' in the target module or theory, where S'_i is the mapping of sort S_i under such a view.

Unmentioned operators also get the identity mapping. Thus, "obvious" parts of a mapping do not need to be explicitly given, namely, any identical mapping of a sort or operator such that its arity and coarity are mapped to those of an operator with the same name in the target can be omitted.[3]

As a first example, the following view `StringAsToset` defines a view from the theory `TOSET`, presented in Section 6.3.1, to the predefined functional module `STRING`, described in Section 7.8.

```
view StringAsToset from TOSET to STRING is
  sort Elt to String .
endv
```

Notice that the identity maps `op _<_ to _<_` and `op _<=_ to _<=_` have been omitted.

The maps sending operators to derived operators, that is, terms with variables, allow us to map an operator, not only to another operator, but also to an expression. The view `RingToRat` below is a view from the theory `RING`, presented in Section 6.3.1, to the predefined functional module `RAT`, described in Section 7.6.

```
view RingToRat from RING to RAT is
  sort Ring to Rat .
  op e to term 1 .
  op z to 0 .
endv
```

Notice that we have followed the convention of omitting the "obvious" parts of the map concerning the operators `_+_` and `_*_`. Furthermore, we have used an operator map sending the operator `e` to the term `1`, due to the fact that in `RAT` `1` is not a constant, but the term `s_^1(0)` (see Sections 4.4.2, 7.2, and 7.6 for details). Note that the map `op e to term 1` cannot be expressed with the other forms of operator maps, because `1` is not an operator, but just syntactic sugar for the term `s_^1(0)`.

As another example, consider the case in which we want to define a view from the theory `NSPOSET` in which we have a sort `Elt` and a non-strict "less or equal" operator `_<=_ : Elt Elt -> Bool`, to a module defining the integers with no such operator but instead with a strict operator "less than" `_<_ : Int Int -> Bool`. Then, we can define a view with maps

```
sort Elt to Int .
op X:Elt <= Y:Elt to term X:Int < Y:Int or X:Int == Y:Int .
```

[3] In Full Maude (see Chapter 21), maps for all sorts in the source theory have to be explicitly given, even when they are identity maps.

where we have also used the predefined equality operator `_==_`. The lefthand side of the operator mapping, `X:Elt <= Y:Elt` in this case, which consists of an operator with only variable arguments, must parse to a unique term in the source theory. Each of the variables used in the maps must have a unique base name (e.g., using both `X:Foo` and `X:Bar` in the same argument list is disallowed).

Also, the righthand side, `X:Int < Y:Int or X:Int == Y:Int` in this case, must parse to a unique term in the target module or theory. The only variables that may occur in the target term are those appearing in the source term; however, they may occur multiple times or not at all. If the source term parses to a sort S or kind $[S]$, then the target term must parse to sort T or kind $[T]$ such that T and the mapping of S under the view S' belong to the same kind.

Views may also contain variable declarations. The syntax is identical to that in modules and theories. However, its semantics is subtly different. Instead of declaring a single variable, a declaration

`var X :` S `.`

now declares two aliases with the same name; in the lefthand side of an operator mapping, `X` is an alias for `X:`S while in the righthand side of an operator mapping, `X` is an alias for `X:`S', with S' the mapping of S under the view.

For example, we can define a view from the theory `SPOSET` with a strict order operation `_<_` to the predefined functional module `INT` of integers (see Section 7.4) in such a way that the `_<_` order relation of a poset is mapped to an expression using the "less than or equal" operator `_<=_` on sort `Int` and the predefined inequality operator `_=/=_` in `BOOL` (see Sections 7.4 and 7.1) as follows:

```
view SPosetToInt from SPOSET to INT is
  sort Elt to Int .
  vars X Y : Elt .
  op X < Y to term X <= Y and X =/= Y .
endv
```

Alternatively, we can specify this view without a variable declaration as

```
view SPosetToInt from SPOSET to INT is
  sort Elt to Int .
  op X:Elt < Y:Elt to term X:Int <= Y:Int and X:Int =/= Y:Int .
endv
```

Note that this view imposes several proof obligations to be checked by the user. In particular, the translations by the view of the axioms in `SPOSET` should hold in the target. Given variables `X`, `Y`, and `Z` of sort `Int`, the following axioms should be true in `INT`:

```
eq X <= X and X =/= X = false .
ceq X <= Z and X =/= Z = true
  if X <= Y and X =/= Y /\ Y <= Z and Y =/= Z .
ceq X = Y if X <= Y and X =/= Y /\ Y <= X and Y =/= X .
```

Of course, since the predefined `INT` module indeed includes both operators `_<_` and `_<=_`, it is not necessary to use the feature described in the previous example. We can instead have simpler view declarations such as the following:

```
view IntAsStoset from STOSET to INT is
  sort Elt to Int .
endv
```

```
view IntAsToset from TOSET to INT is
  sort Elt to Int .
endv
```

where the identity maps `op _<_ to _<_` and `op _<=_ to _<=_` have been omitted.

We recommend following the convention of naming views from `TRIV` by the name of the sort to which `Elt` is mapped, when the name of this sort is not structured.[4] Thus, a view from the theory `TRIV` to the module `INT` that sends the sort `Elt` to `Int` should be named `Int` (as we shall see in Section 7.12.1, the view `Int` is predefined in Maude).

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

This convention can add understandability to the specifications. As we will see in Section 6.3.4, given a module `LIST` of lists parameterized by `TRIV` with a sort `List{X}`, once it is instantiated, e.g., with the view `Int` above, the sort `List{X}` becomes `List{Int}`, defining lists of integers. Using names of views as labels in interfaces of parameterized modules (see Section 6.3.4 below) should be avoided, since this can sometimes generate ambiguities.

We can also have views between theories, which is particularly useful to compose instantiations of views to link the formal parameter of some parameterized module to some actual parameter via some intermediate formal parameter of another parameterized module. We will discuss the uses of these views and give some examples in the coming sections. An example of a view whose target is a theory is the following:

```
view PosetToToset from POSET to TOSET is
  sort Elt to Elt .
endv
```

As said above, identity maps can be omitted. Thus, the following definition is equivalent to the previous one.

```
view PosetToToset from POSET to TOSET is
endv
```

In this example the `PosetToToset` view represents the inclusion of the `POSET` theory into `TOSET`.

In those cases in which a view defines a theory inclusion from `TRIV` into another theory, we recommend following the convention of naming the view with the name of the target theory. An example that will be very useful later is the inclusion of `TRIV` into `TOSET`, which is expressed as a view as follows:

```
view TOSET from TRIV to TOSET is
endv
```

Let us finish this section by commenting on some subtle issues that can arise with operator mappings:

- Operator mappings are not applied to operators that have at least one declaration in a module (as opposed to a theory); if a mapping applies to such an operator, an advisory is generated. Although it does not seem to be useful, Maude does not forbid having subsort-overloaded operators appearing in a theory and in one of its submodules. However, the operator is considered to "belong" to the module, and therefore it cannot be mapped by a view.

[4] Notice that a structured sort name, such as `List{Nat}` for example, cannot be used as a view name, because it is not a single identifier; if desired, the user can write the single-identifier form `List‘{Nat‘}` as view name. The convention is totally general in Full Maude; see Section 6.3.7.

- `assoc` *operators.* Nested occurrences of associative operators may have been flattened, or have been entered in a flattened way such as, for example, `f(a, a, b, b)`. In order to map this to an operator that has different attributes (perhaps including `assoc`) or to a term, flattened occurrences will be temporarily unflattened into a regular term before translation. The precise choice of unflattening is left unspecified.

- `iter` *operators.* Mapping an `iter` operator (see Section 4.4.2) to a non-`iter` operator causes the efficient representation of towers of symbols to be expanded out, with a potential exponential blow up. Mapping an `iter` operator to a term in which the single argument variable occurs more than once causes a doubly exponential blow up. Maude operates under the principle of "you asked for it, you got it" and if the expansion is too large it will die with a virtual memory exhausted error.

- *Built-in operators.* The built-in operators for holding non-algebraically defined data `StringSymbol`, `FloatSymbol`, and `QuotedIdentifierSymbol` have a special internal representation for their terms, and can only be mapped to operators of identical type.

- *Polymorphic operators.* Polymorphic operators must map to polymorphic operators that are polymorphic on the same arguments. Only generic mappings of the form f `to` f' are considered when mapping polymorphic operators.

6.3.3 Parameterized modules

System modules and functional modules can be parameterized. A parameterized system module has syntax

`mod M{`X_1 `::` T_1 `,` ... `,` X_n `::` T_n`} is ... endm`

with $n \geq 1$. Parameterized functional modules have completely analogous syntax.

The `{`X_1 `::` T_1 `,` ... `,` X_n `::` T_n`}` part is called the *interface*, where each pair X_i `::` T_i is a parameter, and each X_i is an identifier—the *parameter name* or *parameter label*—and each T_i is an expression that yields a theory—the *parameter theory.* Each parameter name in an interface must be unique, although there is no uniqueness restriction on the parameter theories of a module—we can have, e.g., two `TRIV` parameters. The parameter theories of a functional module must be functional theories.

In a parameterized module M, all the sorts and statement labels coming from theories in its interface must be qualified by their names. Thus, given a parameter X_i `::` T_i, each sort S in T_i must be qualified as X_i`$`S, and each label l of a statement occurring in T_i must be qualified as X_i`$`l. In fact, the parameterized module M is flattened as follows. For each parameter X_i `::` T_i, a renamed copy of the theory T_i, called X_i `::` T_i is included. The renaming maps each sort S to X_i`$`S, and each label l of a statement occurring in T_i to X_i`$`l. The renaming percolates down through nested inclusions of theories, but has no effect on importations of modules. Thus, if T_i includes a theory T', when the renamed theory X_i `::` T_i is created and included into M, and the renamed theory X_i `::` T' will also be created and included into X_i `::` T_i.[5] However, the renaming will have no effect on modules imported by either the T_i or T'; for example, if `BOOL` is imported by one of these theories, it is not renamed, but imported in the same way into M.

For example, a parameterized module `PRELIM-SET` with `TRIV` as interface can be defined as follows:

[5] These renamed modules are visible as names when using the `show modules` command (see Section 23.12) and will be shared, but they cannot be referred to directly in module expressions.

```
fmod PRELIM-SET{X :: TRIV} is
  protecting BOOL .
  sorts Set NeSet .
  subsorts X$Elt < NeSet < Set .
  op empty : -> Set .
  op _,_ : Set Set -> Set [assoc comm id: empty] .
  op _,_ : NeSet NeSet -> NeSet [ditto] .
  op _in_ : X$Elt Set -> Bool .
  op _-_ : Set Set -> Set . *** set difference

  var E : X$Elt .
  vars S S' : Set .
  eq E, E = E .
  eq E in E, S = true .
  eq E in S = false [owise] .
  eq (E, S) - (E, S') = S - (E, S') .
  eq S - S' = S [owise] .
endfm
```

In Maude—unlike OBJ3 and other similar languages—sorts are not systematically qualified by their module name. This convention of not qualifying sorts may be particularly weak when dealing with parameterized modules. However, given that Maude supports ad-hoc overloading and that constants can be qualified in order to be disambiguated (see Section 3.9.3), the problem of ambiguity in a signature is reduced to collisions of sorts. For example, in a module one may very easily need sets of integers and sets of quoted identifiers, in which case, given the specification of the `PRELIM-SET` module above, we would get two `Set` sorts from different importations which would be confused into one sort. Our solution consists in *qualifying parameterized sorts*, not with the module expression they belong to, but *with the name of the view or views used in the instantiation* of the parameterized module. Since we assume that all views are named, these names are the ones used in the qualification. Specifically, in the body of a parameterized module $M\{X_1 :: T_1 , \ldots , X_n :: T_n\}$, any sort S can be written in the form $S\{X_1 , \ldots , X_n\}$. When the module is instantiated with views $V_1 \ldots V_n$, then this sort is instantiated to $S\{V_1 , \ldots , V_n\}$.

Note that, although we strongly recommend it, the parameterization of sorts is optional, and therefore, for example, the above `PRELIM-SET` specification is a perfectly valid parameterized module.

Sorts declared in the parameterized module $M\{X_1 :: T_1 , \ldots , X_n :: T_n\}$ may in general be parameterized as $S\{Y_1 , \ldots , Y_m\}$, with $m \geq 1$, and where each Y_j is an X_i. It is recommended that all sorts declared in a parameterized module be parameterized with $m = n$ and $Y_j = X_j$ for $1 \leq j \leq n$, but this is not enforced—parameterized sorts may duplicate, omit, or reorder parameters and unparameterized sorts are also allowed.

Thus, the previous `PRELIM-SET` module to define sets could instead have been specified in a better way as follows:

```
fmod BASIC-SET{X :: TRIV} is
  protecting BOOL .
  sorts Set{X} NeSet{X} .
  subsorts X$Elt < NeSet{X} < Set{X} .
  op empty : -> Set{X} .
  op _,_ : Set{X} Set{X} -> Set{X} [assoc comm id: empty] .
  op _,_ : NeSet{X} NeSet{X} -> NeSet{X} [ditto] .
  op _in_ : X$Elt Set{X} -> Bool .
  op _-_ : Set{X} Set{X} -> Set{X} . *** set difference
```

```
  var E : X$Elt .
  vars S S' : Set{X} .
  eq E, E = E .
  eq E in E, S = true .
  eq E in S = false [owise] .
  eq (E, S) - (E, S') = S - (E, S') .
  eq S - S' = S [owise] .
endfm
```

When this module is instantiated with the predefined view `Int`, the sort `Set{X}` becomes `Set{Int}`, and when it is instantiated with the predefined view `Qid` (see Section 7.12.1) it becomes `Set{Qid}`. In the following sections we will see additional examples of how this qualification convention for the sorts of a parameterized module avoids many unintended collisions of sort names, thus making renaming practically unnecessary.[6]

As another simple example of parameterized module, we consider a module `MAYBE{X :: TRIV}` in which we declare a sort `Maybe{X}` as a supersort of the sort `Elt` of the parameter theory and a constant `maybe` of this sort `Maybe{X}`. This technique is useful to declare a partial function as a total function, as we will see in the `PFUN` module of Section 6.3.7.

```
fmod MAYBE{X :: TRIV} is
  sort Maybe{X} .
  subsort X$Elt < Maybe{X} .
  op maybe : -> Maybe{X} [ctor] .
endfm
```

The `PRELIM-SET`, `BASIC-SET`, and `MAYBE` modules above have only one parameter. In general, however, parameterized modules can have several parameters. It can furthermore happen that several parameters are declared with the same parameter theory, that is, we can have, for example, an interface of the form `{X :: TRIV, Y :: TRIV}` involving two copies of the theory `TRIV`. Therefore, parameters cannot be treated as normal submodules, since we do not want them to be shared when their labels are different. We regard the relationship between the body of a parameterized module and the interface of its parameters not as an inclusion, but as a module constructor which is evaluated generating renamed copies of the parameters, which are then included. For the above interface, two copies of the theory `TRIV` are generated, with names `X :: TRIV` and `Y :: TRIV`. As already mentioned, in such copies of parameter theories, sorts are renamed as follows: if Z is the label of a parameter theory T, then each sort S in T is renamed to $Z\$S$ and each statement label l is renamed to $Z\$l$. All occurrences of these sorts and labels in the body of the parameterized module must mention their corresponding renaming.

Let us consider as an example the following module `PAIR`, in which we would like to point out the use of the qualifications for the sorts coming from each of the parameters.

```
fmod PAIR{X :: TRIV, Y :: TRIV} is
  sort Pair{X, Y} .
  op <_;_> : X$Elt Y$Elt -> Pair{X, Y} .
  op 1st : Pair{X, Y} -> X$Elt .
  op 2nd : Pair{X, Y} -> Y$Elt .

  var A : X$Elt .
  var B : Y$Elt .
  eq 1st(< A ; B >) = A .
```

[6] In Section 6.3.7, we shall see how this naming convention can be easily extended to the case of *parameterized views*.

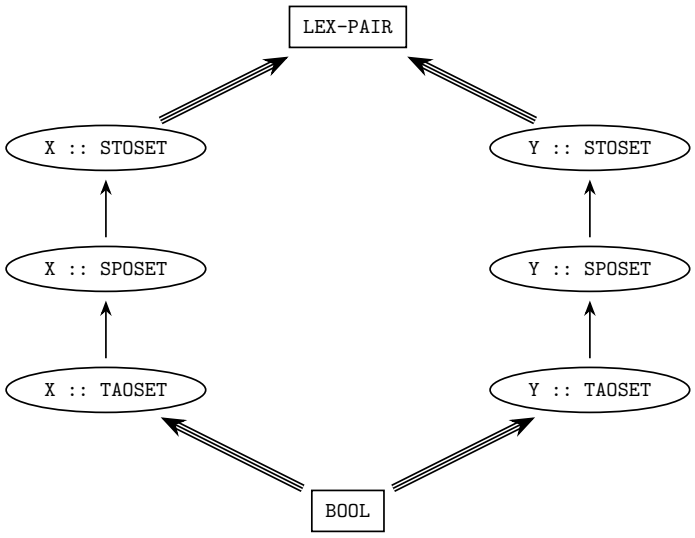


Figure 6.2: Structure of LEX-PAIR

```
  eq 2nd(< A ; B >) = B .
endfm
```

As already mentioned, if a parameter theory is structured, this renaming process for parameter theories is carried out not only at the top level, but for the whole "theory part," that is, renaming *subtheories* but not renaming submodules. Consider, for example, the following parameterized module defining a lexicographical ordering on pairs of elements of two totally strictly ordered sets.

```
fmod LEX-PAIR{X :: STOSET, Y :: STOSET} is
  sort Pair{X, Y} .
  op <_;_> : X$Elt Y$Elt -> Pair{X, Y} .
  op _<_ : Pair{X, Y} Pair{X, Y} -> Bool .
  op 1st : Pair{X, Y} -> X$Elt .
  op 2nd : Pair{X, Y} -> Y$Elt .

  vars A A' : X$Elt .
  vars B B' : Y$Elt .
  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .
  eq < A ; B > < < A' ; B' > = (A < A') or (A == A' and B < B') .
endfm
```

Representing by boxes the modules (with initiality constraints), by ovals the theories (with loose semantics), by triple arrows the `protecting` and parameter importations, and by single arrows the `including` importations, we can depict the structure of the `LEX-PAIR` functional module defining a lexicographic order on pairs as in Figure 6.2, where we have two copies not only of `STOSET` but also of the `SPOSET` and `TAOSET` subtheories (see also Figure 6.1 in page 119), but only one copy of the `BOOL` submodule.

The parameter theory of a module can be any module expression whose result is a theory. The following module defines bags of elements with an `occurrences` operation that returns the

number of occurrences of a particular element in a given bag.

```
fmod BAG{X :: TRIV * (sort Elt to Element)} is
  protecting NAT .
  sorts Bag{X} NeBag{X} .
  subsorts X$Element < NeBag{X} < Bag{X} .
  op mt : -> Bag{X} .
  op __ : Bag{X} Bag{X} -> Bag{X} [assoc comm id: mt] .
  op __ : Bag{X} NeBag{X} -> NeBag{X} [ditto] .
  op occurrences : X$Element Bag{X} -> Nat .

  vars E E' : X$Element .
  var  S : Bag{X} .
  eq occurrences(E, E S) = 1 + occurrences(E, S) .
  eq occurrences(E, S) = 0 [owise] .
endfm
```

Module instantiation will be explained in the next section, and then we shall see some execution examples.

6.3.4 Module instantiation

Instantiation is the process by which actual parameters are bound to the formal parameters of a parameterized module and a new module is created as a result. This can be seen in fact as the evaluation of a module expression. The instantiation requires a view from each formal parameter to its corresponding actual parameter. Each such view is then used to bind the names of sorts, operators, etc. in the formal parameters to the corresponding sorts, operators (or expressions), etc. in the actual target.

The instantiation of a parameterized module must be made with views explicitly defined previously. Thus, given the views `Int` (from `TRIV` to `INT`) and `IntAsStoset` (from `STOSET` to `INT`), both introduced in Section 6.3.2, we can define sets of integers with the module expression `BASIC-SET{Int}`, and lexicographically ordered pairs of integers with `LEX-PAIR{IntAsStoset, IntAsStoset}`.

As mentioned in Section 6.3.2, there are also views from theories to theories. Using such views we can, for example, instantiate the module `BASIC-SET` with the view `TOSET` (from `TRIV` to `TOSET`) given also in Section 6.3.2. The result is a module `BASIC-SET{TOSET}` which is still parameterized, but now by the theory `TOSET`. We can instantiate it again with a view from `TOSET` to some other theory or module, for example, `IntAsToset` (from `TOSET` to `INT`), obtaining the module `BASIC-SET{TOSET}{IntAsToset}`, which defines sets of integers. Note that certain new operations, which would not be meaningful in the original `BASIC-SET` module, could now be defined in a totally parametric way in an extension of `BASIC-SET{TOSET}`. For example, we could define in this way a maximum function

```
op max : NeSet{TOSET}{X} -> X$Elt .
```

as done in the `SET-MAX` module later in this section.

Another interesting use of parameterized modules is the *linking of parameters*. Suppose that we wish to define lists of sets of elements. We may define a module `SET-LIST` parameterized by the theory `TRIV` that imports the module `BASIC-SET` and declares the sort `SetList{X}` with constructors `nil` and `_;_`. Note however that `BASIC-SET` is also a parameterized module, which must be instantiated to be imported. In cases like this one, we can use the label of the parameter to *link* the parameter of the module with the parameter of the submodule. Once the module is instantiated, the parameterized submodule gets instantiated with the same view.

Thus, if the module SET-LIST below is instantiated by, say, the view Int to define lists of sets of integers, the submodule BASIC-SET also gets instantiated with the same view, providing a definition of sets of integers.[7]

```
fmod SET-LIST{X :: TRIV} is
  protecting BASIC-SET{X} .
  sort SetList{X} .
  subsort Set{X} < SetList{X} .
  op nil : -> SetList{X} [ctor] .
  op _;_ : SetList{X} SetList{X} -> SetList{X}
      [ctor assoc id: nil] .
endfm
```

As another example, let us consider the following modules MONOMIAL and POLYNOMIAL, defining, respectively, monomials on a set of variables and polynomials on a commutative ring and a set of variables. First, the module MONOMIAL defines monomials as terms of the form X ^ N, with X a variable[8] and N a nonzero natural number indicating the power to which the variable is raised, and with an *empty syntax* multiplication operation __ on monomials.

```
fmod MONOMIAL{X :: TRIV} is
  protecting NAT .
  sorts Pow{X} Mon{X} .
  subsorts Pow{X} < Mon{X} .
  *** multiplication
  op __ : Mon{X} Mon{X} -> Mon{X} [assoc comm] .
  op _^_ : X$Elt NzNat -> Pow{X} .
  var  X : X$Elt .
  vars N M : NzNat .
  eq (X ^ N) (X ^ M) = X ^ (N + M) .
endfm
```

Once we have the specification of monomials, we can specify polynomials as monomials with coefficients in some commutative ring, and with addition and multiplication operations. Thus, for specifying polynomials on a ring and a set of variables in a module POLYNOMIAL, we need to import the above module MONOMIAL. But notice that POLYNOMIAL is parameterized by *two* theories: RING, for the coefficients, and TRIV, for the variables. Since we need to import monomials on the *same* set of variables, we need to *bind* or *link* such parameters. This linking is done by means of the label X of the parameter theory X :: TRIV.

```
fmod POLYNOMIAL{R :: RING, X :: TRIV} is
  protecting MONOMIAL{X} .
  sorts Poly{R, X} .
  subsorts R$Ring < Poly{R, X} .
  *** multiplication
  op __ : Poly{R, X} Poly{R, X} -> Poly{R, X} [assoc comm] .
  *** addition
  op _++_ : Poly{R, X} Poly{R, X} -> Poly{R, X} [assoc comm] .
  op --_ : Poly{R, X} -> Poly{R, X} .
```

[7] In Section 6.3.7, we shall introduce the notion of *parameterized views*, a more convenient way of defining this kind of structures.

[8] Note that a variable in a monomial or polynomial is a *constant*, *not* a mathematical variable in the Maude sense. That is, in this example variables are understood as *names*. Of course, in Maude we can also define a variable X:X$Elt in the parameter sort to which variables belong as constants, or, more generally, variables such as P:Poly{R, X}. In this context such mathematical variables can be distinguished from variables as names by referring to them as *metavariables*.

```
  op __ : R$Ring Mon{X} -> Poly{R, X} .

  vars A B : R$Ring .
  vars U V : Mon{X} .
  vars P Q R : Poly{R, X} .
  eq P ++ z = P .
  eq P ++ (-- P) = z .
  eq P e = P .
  eq -- (P ++ Q) = (-- P) ++ (-- Q) .
  eq -- (A U) = (- A) U .
  eq P (Q ++ R) = (P Q) ++ (P R) .
  eq z U = z .
  eq z P = z .
  eq A (B U) = (A B) U .
  eq (A U) ++ (B U) = (A ++ B) U .
  eq (A U) (B V) = (A B) (U V) .
  eq A B = A * B .
  eq A ++ B = A + B .
endfm
```

If the module `POLYNOMIAL` is instantiated with, say, views `RingToRat` and `Qid`, the submodule `MONOMIAL` then gets automatically instantiated with `Qid`, thanks to the binding of the parameters.

As an additional example, let us give a more concise definition of the parameterized module `LEX-PAIR{X :: STOSET, Y :: STOSET}` given in Section 6.3.3 using these ideas as follows:

```
view STOSET from TRIV to STOSET is
endv

fmod LEX-PAIR{X :: STOSET, Y :: STOSET} is
  protecting PAIR{STOSET, STOSET}{X, Y} .
  op _<_ :
     Pair{STOSET, STOSET}{X, Y} Pair{STOSET, STOSET}{X, Y} -> Bool .
  vars A A' : X$Elt .
  vars B B' : Y$Elt .
  eq < A ; B > < < A' ; B' > = (A < A') or (A == A' and B < B') .
endfm
```

In Section 6.2.2, we presented a `NAT-LIST-MAX` module in which we defined a `max` function that returns the greatest element of a list of natural numbers. However, we can define such a function on lists or sets of any type of elements as long as there is a total order relation available for them. Let us consider the following module `SET-MAX`, parameterized by the theory `TOSET` (see Section 6.3.1. Given a non-empty finite set of elements in a totally ordered set, the operation `max` returns the maximum element in the set. Note that we have used the `or-else` operator for short-circuit disjunction from the `EXT-BOOL` module to improve the efficiency of the calculation.

```
fmod SET-MAX{T :: TOSET} is
  protecting BASIC-SET{TOSET}{T} .
  protecting EXT-BOOL .
  op max : NeSet{TOSET}{T} -> T$Elt .
  var E : T$Elt .
  var S : Set{TOSET}{T} .
  eq max(E, S)
    = if S == empty or-else max(S) < E
```

```
      then E
      else max(S)
      fi .
endfm
```

We can now calculate the maximum of a set of integers by instantiating this module with the view `IntAsToset` introduced in Section 6.3.2. Notice that in this example we need an extra set of parentheses to disambiguate between the operator `max` just defined and the associative operator `max` on integers.

```
fmod INT-SET-MAX is
  protecting SET-MAX{IntAsToset} .
endfm

  Maude> red max((4, 3, 5, 2, 1)) .
  result NzNat: 5
```

Similarly, we can calculate the greatest element in sets of any type with a total order relation; for example, sets of strings, by using the view `StringAsToset` also introduced in Section 6.3.2:

```
fmod STRING-SET-MAX is
  protecting SET-MAX{StringAsToset} .
endfm

Maude> red max("four", "three", "five", "two", "one") .
result String: "two"
```

Notice that, if we have several parameters, we can instantiate the parameterized module or theory with some views going to theories and others going to modules. The result in such case is the expected one, that is, we get a module or theory parameterized by the targets of those views going to theories. For example, the module `RAT-POLY` below gives us a specification of the polynomials with rational coefficients by just importing the module `POLYNOMIAL` introduced above instantiated with the view `RingToRat` from the theory `RING` to the functional module `RAT` (see Section 6.3.2).

```
fmod RAT-POLY{X :: TRIV} is
  protecting POLYNOMIAL{RingToRat, X} .
endfm
```

We can then define the polynomials with rational coefficients and with quoted identifiers as variables by instantiating the module `RAT-POLY` with the following `Qid` view, which is predefined in Maude (see Section 7.12.1).

```
view Qid from TRIV to QID is
  sort Elt to Qid .
endv

fmod QID-RAT-POLY is
  pr RAT-POLY{Qid} .
endfm
```

Let us reduce as an example the following polynomial expression:

```
Maude> red in QID-RAT-POLY :
        (((2 / 3) (('X ^ 2) ('Y ^ 3)))
         ++ ((7 / 5) (('Y ^ 2) ('Z ^ 5))))
        (((1 / 7) ('U ^ 2))
```

```
          ++ (1 / 2)) .
result Poly{RingToRat, Qid}:
           (1/3 ('X ^ 2) 'Y ^ 3)
         ++ (1/5 ('U ^ 2) ('Y ^ 2) 'Z ^ 5)
         ++ (2/21 ('U ^ 2) ('X ^ 2) 'Y ^ 3)
         ++ (7/10 ('Y ^ 2) 'Z ^ 5)
```

Summarizing, a parameterized module $M\{X_1 :: T_1 , \ldots , X_n :: T_n\}$ with n free parameters is instantiated by the module expression $M\{A_1 , \ldots , A_n\}$, where each A_i is an instance of one of the following three alternatives:

- The name Y_j of a parameter of the correct theory from the module enclosing the module expression. In this case the parameter becomes a *bound parameter* in the module resulting from the instantiation. Each sort $X_i\$S$ is mapped to $Y_j\$S$, and each X_i occurring as a parameter in a parameterized sort becomes Y_j (and similarly for statement labels).
- The name of a view V with a theory as target with the correct source theory. In this case, the parameter becomes a *free parameter* with V's target theory in the module resulting from the instantiation.
- The name of a view V with a module as target with the correct source theory. In this case, *the parameter disappears*. Each sort $X_i\$S$ is mapped to S', where S' is the mapping of S under V. Each X_i occurring as a parameter in a parameterized sort becomes V. Each statement label $X_i\$l$ is mapped to l', where l' is the mapping of l under the view V.

Parameterized modules with free parameters cannot be imported: first all of the free parameters must be instantiated away. Parameterized modules with bound parameters may only be imported, since they were created for module expressions in a context where the parameters are bound by an enclosing parameterized module.

Parameterized functional modules may be instantiated with views that have system modules as their targets; then the instantiated module is promoted to a system module.

Parameterized modules cannot be summed, even if all the parameters are bound. Parameterized modules may be renamed, but the renaming must not affect any sorts or operators coming from a parameter theory. The result of renaming a parameterized module is a parameterized module with the same parameters, and we can use it as any other parameterized module; for example, we can instantiate it with a view, or bind its parameters to the parameters of the module in which the module expression is being imported, as in the following example, where we rename the `SET-LIST` parameterized module above.

```
fmod MY-SET-LIST{Y :: TRIV} is
  pr (SET-LIST
       * (sort Set{X} to MySet{X},
          op __ : SetList{X} SetList{X} -> SetList{X} to _._))
       {Y} .
endfm

fmod MY-QID-SET-LIST is
  protecting MY-SET-LIST{Qid} .
endfm
```

The `SET-LIST` module has only free parameters and so it can be renamed; however its renaming imports the renaming of `BASIC-SET{X}` which has a bound parameter. Note that the parameter of the sorts appearing in the renaming of the `SET-LIST` module is `X`, since this is the label of

the parameter in such module. We have used label `Y` for the parameter of `MY-SET-LIST` to emphasize this fact, although they could be the same.

Allowing renaming of modules with bound parameters requires that renamings be capable of instantiation; that is, parameterized sort names inside a renaming have their parameters instantiated, with an extra pair of curly brackets being added in the case of instantiation by a view with a theory as target.

Let us illustrate these ideas. When, due to instantiation by a view with a theory as target, a bound parameter in a renamed module escapes and needs to be rebound by an extra instantiation, the extra instantiation is inserted *before* rather than after the renaming. Let us consider the following example, where we use the views `TOSET`, from the theory `TRIV` to the theory `TOSET`, and `IntAsToset`, from the theory `TOSET` to the predefined module `INT`, both described in Section 6.3.2.

```
fmod RENAMING-PAR-MOD-A{X :: TRIV} is
  sort Foo{X} .
  op f : Foo{X} -> Foo{X} .
endfm

fmod RENAMING-PAR-MOD-B{X :: TRIV} is
  extending RENAMING-PAR-MOD-A{X} .
  sort Bar{X} .
  op g : Bar{X} -> Foo{X} .
endfm

fmod RENAMING-PAR-MOD-C is
  pr (RENAMING-PAR-MOD-B * (sort Foo{X} to Foo'{X},
            sort Bar{X} to Bar'{X},
            op f : Foo{X} -> Foo{X} to f',
            op g : Bar{X} -> Foo{X} to g')) {TOSET} {IntAsToset} .
endfm
```

In this case, the module `RENAMING-PAR-MOD-A` gets instantiated before it is renamed:

```
RENAMING-PAR-MOD-A{TOSET}{IntAsToset}
  * (sort Foo{TOSET}{IntAsToset} to Foo'{TOSET}{IntAsToset},
     op f : [Foo{TOSET}{IntAsToset}] -> [Foo{TOSET}{IntAsToset}]
       to f')
```

Passing parameters from an enclosing module in nonfinal instantiations is prohibited. This restriction avoids many subtle issues. Thus:

```
fmod ILLEGAL-INST{X :: RING, Y :: POSET} is
  protecting POLYNOMIAL{X, POSET}{Y} .
endfm
```

is *illegal*, because `X` occurs in the nonfinal instantiation `POLYNOMIAL{X, POSET}`. With appropriate views, this example can be correctly written as follows:

```
view RING from RING to RING is
endv

view POSET from TRIV to POSET is
endv

fmod LEGAL-INST{X :: RING, Y :: POSET} is
  protecting POLYNOMIAL{RING, POSET}{X, Y} .
endfm
```

Another way of viewing this restriction is that parameters from an enclosing module and views with theories as targets may not occur in the same instantiation. Note that views with theories as targets may never occur in a final instantiation (otherwise there would be free parameters in an import) and must occur in any nonfinal instantiation (otherwise there would be no free parameters for the next instantiation).

6.3.5 Lists

There are different ways of building lists. One possibility is to begin with the empty list and the singleton lists, and then use the concatenation operation to get bigger lists. However, concatenation cannot be a free list constructor, because it satisfies an associativity equation and has the empty list as identity. This approach will be used in the predefined module for generic lists described in Section 7.13.1, and appears in many similar examples throughout this book. Given the support for equational attributes (associativity, commutativity, etc.) in Maude, as explained in Section 4.4.1, one can argue that this is indeed the most natural specification for lists in Maude.

Here we use instead the two standard free constructors for lists that can be found in many functional programming languages: the empty list *nil*, here denoted `[]`, and the *cons* operation that adds an element to the beginning of a list, here denoted with the mixfix syntax `_:_`. This approach facilitates proving properties of lists by structural induction in Maude's inductive theorem prover (ITP), and provides a simple basis for specifying sorted lists and sorting operations on them in Section 6.3.6.

As usual, `head` and `tail` are the selectors associated with the `_:_` constructor. Since they are not defined on the empty list, we avoid their partiality in the same way as we have done for stacks and queues in the previous sections by means of a subsort `NeList` of non-empty lists.

```
fmod LIST-CONS{X :: TRIV} is
  protecting NAT .

  sorts NeList{X} List{X} .
  subsort NeList{X} < List{X} .

  op [] : -> List{X} [ctor] .
  op _:_ : X$Elt List{X} -> NeList{X} [ctor] .
  op tail : NeList{X} -> List{X} .
  op head : NeList{X} -> X$Elt .

  var E : X$Elt .
  var N : Nat .
  vars L L' : List{X} .

  eq tail(E : L) = L .
  eq head(E : L) = E .
```

Three interesting operations on lists are list concatenation (here denoted with mixfix syntax `_++_`), the length of a list, and reversing a list. The `length` operator has a result of sort `Nat`, that comes from the predefined module `NAT`, imported in `protecting` mode. These three operations are defined as usual by structural induction on the two constructors, with an equation for the empty base case and another for the *cons* case `E : L`.

Here we are not concerned with efficiency and therefore we just specify the operations in a simple way, without using, for example, tail-recursive auxiliary operations in the style of Section 7.13.1.

```
  op _++_ : List{X} List{X} -> List{X} .
  op length : List{X} -> Nat .
  op reverse : List{X} -> List{X} .

  eq [] ++ L = L .
  eq (E : L) ++ L' = E : (L ++ L') .
  eq length([]) = 0 .
  eq length(E : L) = 1 + length(L) .
  eq reverse([]) = [] .
  eq reverse(E : L) = reverse(L) ++ (E : []) .
```

In this specification of generic lists we also add two operations that will be useful later, in Section 6.3.6, when sorting lists: `take_from_` and `throw_from_`. The first one builds a list by taking the first n elements of the given list, while the second one deletes the first n elements of the given list. Both of them are defined by structural induction on both arguments, the base case being when either the first is 0 or the second is empty. As usual, `s_` denotes the successor operator on natural numbers.

```
  op take_from_ : Nat List{X} -> List{X} .
  op throw_from_ : Nat List{X} -> List{X} .

  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
```

The following sample reduction shows the result of reversing a list of character strings.

```
fmod LIST-CONS-TEST is
  protecting LIST-CONS{String} .
endfm

Maude> red reverse("one" : "two" : "three" : []) .
result NeList{String}: "three" : "two" : "one" : []
```

6.3.6 Sorted lists

In order-sorted equational specifications, subsorts must be defined by means of constructors, but it is not possible to have a subsort of sorted lists, for example, defined by a property over lists; a more expressive formalism is needed. *Membership equational logic* allows subsort definition by means of conditions involving equations and/or sort predicates. In this example we use this technique to define a subsort of *sorted lists*, included in the sort of lists imported from the module `LIST-CONS` in Section 6.3.5. Furthermore, we will also specify here different well-known sorting algorithms.

Parameterized sorted lists need a stronger requirement than `TRIV`, because a total order over the elements to be sorted is needed. Since repetitions pose no problems for sorting a list, the order relation should be non-strict, like in the `NSTOSET` theory introduced in Section 6.3.1. However, for the specification of the sorting algorithms, it is more convenient to use also the strict version of the order. For these reasons, we will use as requirement for parameterized sorted lists the theory `TOSET`, also introduced in Section 6.3.1.

The parameterized module for sorted lists imports the parameterized list module. However, note that we want lists over a totally ordered set, instead of lists over any set; therefore, first we *partially instantiate* `LIST-CONS` with an inclusion view from the theory `TRIV` to the theory `TOSET`.

```
view TOSET from TRIV to TOSET is
endv
```

We are still left with a parameterized module and corresponding dependent sorts, but now with respect to the `TOSET` requirement. This is the reason justifying the notation `LIST-CONS{TOSET}{X}` in the `protecting` importation below, as well as `NeList{TOSET}{X}` and `List{TOSET}{X}` as names of the imported sorts.

Notice the three (conditional) membership axioms defining the subsort `SortedList{X}`: the empty and singleton lists are always sorted, and a longer list is sorted when the first element is less than or equal to the second, and the list without the first element is also sorted.

```
fmod SORTED-LIST{X :: TOSET} is
  protecting LIST-CONS{TOSET}{X} .

  sorts SortedList{X} NeSortedList{X} .
  subsorts NeSortedList{X} < SortedList{X} < List{TOSET}{X} .
  subsort NeSortedList{X} < NeList{TOSET}{X} .

  vars N M : X$Elt .
  vars L L' : List{TOSET}{X} .
  vars OL OL' : SortedList{X} .
  var NEOL : NeSortedList{X} .

  mb [] : SortedList{X} .
  mb (N : []) : NeSortedList{X} .
  cmb (N : NEOL) : NeSortedList{X} if N <= head(NEOL) .
```

As part of this module, we also define several well-known sorting operations: `insertion-sort`, `quicksort`, and `mergesort`, based on appropriate auxiliary operations. 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 or functional programming languages. For example, `insertion-sort` is declared as an operation from `List{TOSET}{X}` to `SortedList{X}`, instead of the much less informative typing from `List{TOSET}{X}` to `List{TOSET}{X}`. The same applies to each of the auxiliary operations. Furthermore, a function that requires its input argument to be a sorted list can now be defined as a *total* function, whereas in less expressive typing formalisms it would have to be either *partial*, or to be defined with exceptional behavior on the erroneous arguments.

The operation `insert-list` inserts an element in the appropriate position of an already sorted list, so that the resulting list is also sorted. The sorting operation `insertion-sort` recursively sorts the list without the first element and then calls `insert-list`, which inserts the missing element in the correct position.

```
  op insertion-sort : List{TOSET}{X} -> SortedList{X} .
  op insert-list : SortedList{X} X$Elt -> SortedList{X} .

  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 N < M .
```

The sorting operation `mergesort` splits a given list in half by means of the operations `take_from_` and `throw_from_` described in Section 6.3.5 above, recursively sorts each sublist, and then calls the commutative `merge` operation on the sorted sublists to obtain the final sorted result. In Section 7.13.6 on sortable lists there is a more efficient (albeit more complex) definition of the mergesort algorithm on lists.

```
  op mergesort : List{TOSET}{X} -> SortedList{X} .
  op merge : SortedList{X} SortedList{X} -> SortedList{X} [comm] .

  eq mergesort([]) = [] .
  eq mergesort(N : []) = N : [] .
  ceq mergesort(L)
    = merge(mergesort(take (length(L) quo 2) from L),
            mergesort(throw (length(L) quo 2) from L))
    if length(L) > 1 .

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

Finally, `quicksort` works on a list by separating its elements into those smaller than the first element (taken as the pivot) and those bigger than the first, recursively sorts each of the resulting lists, and simply puts them together by concatenating them with the pivot in the middle.

```
  op quicksort : List{TOSET}{X} -> SortedList{X} .
  op leq-elems : List{TOSET}{X} X$Elt -> List{TOSET}{X} .
  op gr-elems : List{TOSET}{X} X$Elt -> List{TOSET}{X} .

  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 M < N .
  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 M < N .
endfm
```

We now apply the sorting operations to lists of natural numbers.

```
view NatAsToset from TOSET to NAT is
  sort Elt to Nat .
endv

fmod SORTED-LIST-TEST is
  protecting SORTED-LIST{NatAsToset} .
endfm

Maude> red insertion-sort(5 : 4 : 3 : 2 : 1 : 0 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []
```

```
Maude> red mergesort(5 : 3 : 1 : 0 : 2 : 4 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []

Maude> red quicksort(0 : 1 : 2 : 5 : 4 : 3 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []
```

6.3.7 Parameterized views

Suppose we have defined modules `LIST{X :: TRIV}` and `SET{X :: TRIV}`, specifying, respectively, lists and sets, and suppose that we need, e.g., the data type of lists of sets of natural numbers. Typically, we would first instantiate the module `SET` with a

view, say `Nat`, from `TRIV` to the module `NAT` mapping the sort `Elt` to the sort `Nat`, thus getting the module `SET{Nat}` of sets of natural numbers. Then, we would instantiate the module specifying lists with a view, say `NatSet`, from `TRIV` to `SET{Nat}`, obtaining the module `LIST{NatSet}`. But, what if we need now the data type of lists of sets of Booleans? Should we repeat the whole process again? One possibility is to define a combined module `SET-LIST{X :: TRIV}`. But what if we later want stacks of sets instead of lists of sets?

We can greatly improve the reusability of specifications by using *parameterized views*. Let us consider the following parameterized vietw `Set` from `TRIV` to `SET`, which maps the sort `Elt` to the sort `Set{X}`.

```
view Set{X :: TRIV} from TRIV to SET{X} is
  sort Elt to Set{X} .
endv
```

With this kind of views we can keep the parameter part of the target module still as a parameter. We can now have lists of sets, stacks of sets, and so on, for any instance of `TRIV`, by instantiating the appropriate parameterized module with the appropriate view. For example, given the view `Nat` above, we can have the module `LIST{Set{Nat}}` of lists of sets of natural numbers, or lists of sets of Booleans with `LIST{Set{Bool}}`, given a view `Bool` from `TRIV` to the predefined module `BOOL`. Similarly, we can have `STACK{Set{Nat}}` or `STACK{Set{Bool}}`.

We can also link the parameter of a module like `LIST{Set{X}}` to the parameter of the module in which it is being included. That is, we can, for example, declare a module of the form

```
fmod GENERIC-SET-LIST{X :: TRIV} is
  protecting LIST{Set{X}} .
endfm
```

Then, instantiating the parameterized module `GENERIC-SET-LIST` with a view `V` from `TRIV` to another module or theory results in a module with name `GENERIC-SET-LIST{V}`, which includes the module `LIST{Set{V}}`. Note that even with parameterized views we still follow conventions for module interfaces and for sort names (see Section 6.3). The only difference is that now, instead of having simple view names, we must consider names of views which are parameterized.

The use of parameterized views in the instantiation of parameterized modules allows very reusable specifications. For example, a very simple way of specifying (finite) partial functions is to see a partial function as a set of input-result pairs.[9] Of course, for such a set to represent a function there cannot be two pairs associating different results with the same input value. We show later in this section (in the module `PFUN` below) how this property can be specified by means of appropriate membership axioms. Note, however, that since membership axioms

[9] An alternative specification is available in Maude's prelude, see Section 7.14.

cannot be given on associative operators over sorts (see Section 20.3.8), we cannot use either the specification of sets described in Section 6.3.3 or the predefined module in Section 7.13.2. Let us consider instead the following module:

```
fmod SET-KIND{X :: TRIV} is
  sorts NeKSet{X} KSet{X} .
  subsort X$Elt < NeKSet{X} < KSet{X} .
  op empty : -> KSet{X} [ctor] .
  op _,_ : KSet{X} KSet{X} ~> KSet{X} [ctor assoc comm id: empty] .
  mb NS:NeKSet{X}, NS':NeKSet{X} : NeKSet{X} .
  eq E:X$Elt, E:X$Elt = E:X$Elt . *** idempotency
endfm
```

Here the operator `_,_` is declared at the kind level—notice the different form of the arrow in its declaration—together with a membership axiom, that is logically equivalent to the declaration

```
op _,_ : NeKSet{X} NeKSet{X} -> NeKSet{X} .
```

at the sort level.

We can then specify sets of pairs by instantiating this `SET-KIND` module with a parameterized view from `TRIV` to the parameterized module `PAIR{X, Y}` defining pairs of elements introduced in Section 6.3.3. The appropriate parameterized view can be defined as follows:

```
view Pair{X :: TRIV, Y :: TRIV} from TRIV to PAIR{X, Y} is
  sort Elt to Pair{X, Y} .
endv
```

A partial function can be lifted to a total function by adding a special value to its codomain, to be used as the result for the input elements for which the function is not defined. For this we make good use of the parameterized module `MAYBE`, introduced in Section 6.3.3, which adds a supersort and a new element `maybe` to this supersort; in this application, the constant `maybe` is renamed to `undefined`.

We are now ready to give the specification of partial functions. The sets representing the domain and codomain of the function are given by `TRIV` parameters, and then the set of tuples is provided by the imported module expression `SET-KIND{Pair{X, Y}}` with sorts `KSet{Pair{X, Y}}` and `NeKSet{Pair{X, Y}}`. We define operations `dom` and `im` returning, respectively, the domain and image of a set of pairs. The `dom` operation will be used for checking whether there is already a pair in a set of pairs with a given input value. With these declarations we can define the sort `PFun{X, Y}` as a subsort of `KSet{Pair{X, Y}}`, by adding the appropriate membership axioms specifying those sets that satisfy the required property. Finally, we define operators `_[_]` and `_[_->_]` to evaluate a function at a particular element, and to add or redefine an input-result pair, respectively. We use the Maude predefined module `SET` (see Section 7.13.2) for representing the sets of elements in the domain and image of a partial function.

```
fmod PFUN{X :: TRIV, Y :: TRIV} is
  pr SET-KIND{Pair{X, Y}} .
  pr SET{X} + SET{Y} .
  pr (MAYBE * (op maybe to undefined)){Y} .

  sort PFun{X, Y} .
  subsorts Pair{X, Y} < PFun{X, Y} < KSet{Pair{X, Y}} .

  vars A D : X$Elt .
  vars B C : Y$Elt .
```

```
  var  F : PFun{X, Y} .
  var  S : KSet{Pair{X, Y}} .

  op dom : KSet{Pair{X, Y}} -> Set{X} .     *** domain
  eq dom(empty) = empty .
  eq dom(< A ; B >, S) = A, dom(S) .
  op im : KSet{Pair{X, Y}} -> Set{Y} .      *** image
  eq im(empty) = empty .
  eq im(< A ; B >, S) = B, im(S) .

  op empty : -> PFun{X, Y} [ctor] .
  cmb < A ; B >, < D ; C >, F : PFun{X, Y}
    if < D ; C >, F : PFun{X, Y} /\ not(A in dom(< D ; C >, F)) .

  op _[_] : PFun{X, Y} X$Elt -> Maybe{Y} .
  op _[_->_] : PFun{X, Y} X$Elt Y$Elt -> PFun{X, Y} .
  ceq (< A ; B >, F)[ A ] = B
    if (< A ; B >, F) : PFun{X, Y} .
  eq F [ A ] = undefined [owise] .
  ceq (< A ; B >, F)[ A -> C ] = < A ; C >, F
    if (< A ; B >, F) : PFun{X, Y} .
  eq F [ A -> C ] = < A ; C >, F [owise] .
endfm
```

Now, we can instantiate the `PFUN` module with, for example, views `Set{String}` and `Nat`, in order to get the finite partial functions from string sets to natural numbers by means of the module expression `PFUN{Set{String}, Nat}`.

```
fmod SIZES is
  pr PFUN{List{String}, Nat} .
endfm
```

Notice that views can be instantiated directly with another view rather than a parameter from an enclosing module or view. We can then do some rewrites:

```
Maude> red in SIZES :
        (< "o" ; 1 >, < "o" "t" ; 2 >, < "o" "t" "t" ; 3 >) ["o" "t"] .
result NzNat: 2
```

View nesting may be useful in some occasions. For instance, suppose that we want to generalize our `SIZES` module so that lists can be of any type and we want to allow default values. The `MAYBE` module in Section 6.3.3 is handy for this.

```
view Maybe{X :: TRIV} from TRIV to MAYBE{X} is
  sort Elt to Maybe{X} .
endv
```

To allow lists of any type, including `null` elements, we can specify the following module.

```
fmod NULL-SIZES{X :: TRIV} is
  pr PFUN{List{Maybe{X}}, Nat} * (op maybe to null) .
endfm

fmod STRING-NULL-SIZES is
  pr NULL-SIZES{String} .
endfm
```

```
Maude> red in STRING-NULL-SIZES :
       (< "o" null ; 1 >, < null "o" "t" ; 2 >, < "o" null "t" "t" ; 3 >) ["o" "t"] .
result Maybe{Nat}: undefined

Maude> red in STRING-NULL-SIZES :
        (< "o" null ; 1 >, < null "o" "t" ; 2 >, < "o" null "t" "t" ; 3 >) [null "o" "t"] .
result NzNat: 2
```

Parameterized views can also be instantiated on theory-views to change the theory of parameters. Assume we wanted to have our sizes structure on lists of elements with a total order. We could use predefined theory-views `STRICT-WEAK-ORDER` and `STRICT-TOTAL-ORDER` as follows:

```
fmod SORTED-SIZES{X :: STRICT-TOTAL-ORDER, Y :: TRIV} is
  pr PFUN{List{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}, Y} .
endfm
```

And then use predefined views `Nat<` and `String` to instantiate it.

```
fmod NAT-SORTED-SIZES is
  pr SORTED-SIZES{Nat<, String} .
endfm

Maude> red in NAT-SORTED-SIZES :
        (< 0 ; "zero" >, < 1 ; "one" >, < 2 ; "two" >) [1] .
result String: "one"
```

Just as with module instantiation, a view instantiation that includes a theory-view is nonfinal since the parameter taking the theory-view must be recaptured or instantiated in a final instantiation. Note that as with module instantiations, bound parameters may not be passed in a nonfinal instantiation. Furthermore for both module instantiations and view instantiations, nested view instantiations with bound parameters may not be passed in a nonfinal instantiation. In other words view instantiations with bound parameters must obey the same rule as bare bound parameters.

One subtle point is that an operator-to-term mapping in a parameterized view may use any operators in the (parameterized) to-module in its to-term, including operators from parameter-theories. Thus by extension, operator-to-operator mappings may map operators in the from-theory to operators in a parameter-theory and sort-to-sort mappings may map sorts in a from-theory to sorts in a parameter-theory:

```
fmod PAR-TH-EXAMPLE{X :: TRIV} is
  sort Foo{X} .
  op f : X$Elt -> Foo{X} .
endfm

view PAR-TH-EXAMPLE{X :: TRIV} from TRIV to PAR-TH-EXAMPLE{X} is
  sort Elt to X$Elt . *** mapping from-theory sort to parameter-theory sort
endv
```

Having parameterized views allows complex nesting of instantiations with bound parameters which can be difficult or impossible to evaluate correctly when the bound parameters are instantiated. Thus, two technical restrictions are enforced on the use of parameterized views:

- *Every parameter of a parameterized view must appear as a bound parameter in its to-module.* If a parameter does not appear in the to-module then it is effectively useless and should be removed.

- We introduce the notion of a conflict between parameters: Two bound parameters, `X` and `Y` are in conflict if `X` is an argument to a module or view final-instantiation, and `Y` is a bound parameter in some view expression that is also an argument in the same final-instantiation.

  For example, the module expressions `M2{X, V{Y}}` and `M{V2{X, V{Y}}}` both induce a conflict on bound parameters `X` and `Y`. Conflicts on bound parameters in a module expression are propagated to the free parameters of a view that uses that module expression as its to-module and to the free parameters of a module that imports that module expression. So, for example:

  ```
  view V3{X :: TRIV, Y :: TRIV} from TRIV to M{X, V{Y}} is
    ...
  endv
  ```

  receives a conflict between its `X` and `Y` parameters. Conflicts are also propagated when parameters are directly instantiated by parameters from an enclosing module or view. So, for example,

  ```
  fmod M3{A :: TRIV, B :: TRIV} is
    inc M{V3{A, B}} .
    ...
  endfm
  ```

  receives a conflict between its `A` and `B` parameters because they are used to instantiate conflicting parameters in `V3`.

  A parameter may have a conflict with itself. For example,

  ```
  fmod MMAP{X :: TRIV} is
    inc MAP{X, Set{X}} .
  endfm
  ```

  For the most part, conflicts are quietly generated and propagated through arbitrarily deep nesting of importations and instantiations without users being aware of them. The only place they become important is in nonfinal instantiations, when the following restriction is enforced: *The nonfinal instantiation of a module or view may not pass theory-view arguments to two parameters that have a conflict (or to a parameter that has a conflict with itself).* Passing a theory-view to one of a pair of conflicting parameters is allowed as long as the other receives a (unparameterized) module-view (recall bound parameters are not allowed in nonfinal instantiations). This restriction avoids situations where the recapture of a bound parameter following instantiation by a theory-view results in the bound parameter appearing in a nonfinal instantiation. For example, if `X` and `Y` were instantiated by theory-views `TV1` and `TV2`, respectively, then the expression `M{V2{X, V{Y}}}` would become `M{V2{TV1, V{TV2}{Y}}{X}}`, and `Y` appears as a bound parameter in a nonfinal instantiation of view `V2`: `V2{TV1, V{TV2}{Y}}`.

Chapter 7

Predefined Data Modules

Maude has a standard library of predefined modules that, by default, are entered into the system at the beginning of each session, so that any of these predefined modules can be imported by any other module defined by the user. Also, by default, the predefined functional module `BOOL` is automatically imported (in `including` mode) as a submodule of any user-defined module, unless such importation is explicitly disabled. These modules can be found in the file `prelude.maude` that is part of the Maude distribution.

We describe below those predefined modules that provide commonly used data types, including Booleans, numbers, strings, and quoted identifiers. The relationships among these modules are shown in the importation graph in Figure 7.1, where all the importations are in `protecting` mode.

We also describe typical *parameterized* collections of data types such as lists and sets, and associations such as maps and arrays. The chapter ends introducing the built-in linear Diophantine equation solver, defined in the file `linear.maude` that is also part of the Maude distribution.

Other predefined modules, also in the `prelude.maude` file, are discussed later; more specifically, the `META-LEVEL` module is discussed in Chapter 17, the `LOOP-MODE` module in Section 18.4, and the `CONFIGURATION` module in Sections 8.1 and 9.

Furthermore, this chapter also describes a predefined module `MACHINE-INT` for machine integers, which is obtained from the module `INT` of (arbitrary size) integers, but is distributed in a separate file `machine-int.maude`.

As explained in Section 4.4.10, many operators in predefined modules are declared with the `special` attribute, so that they are to be treated as *built-in* operators associated with appropriate C++ code by "hooks" specified after the `special` attribute. In what follows, to lighten the exposition, we will omit the details about such hooks in special operators, writing `special (...)` instead. The full definitions can be found in the file `prelude.maude`.

Most built-in data types are algebraically constructed, that is, they are built out of constants and constructor operators; however, floating point numbers (floats), strings, and quoted identifiers (qids) are treated as countable sets of constants and are represented by "special" operators `<Floats>`, `<Strings>`, and `<Qids>`, respectively. These operators are used in specifying the hooks mentioned above, but they cannot be used explicitly in terms.

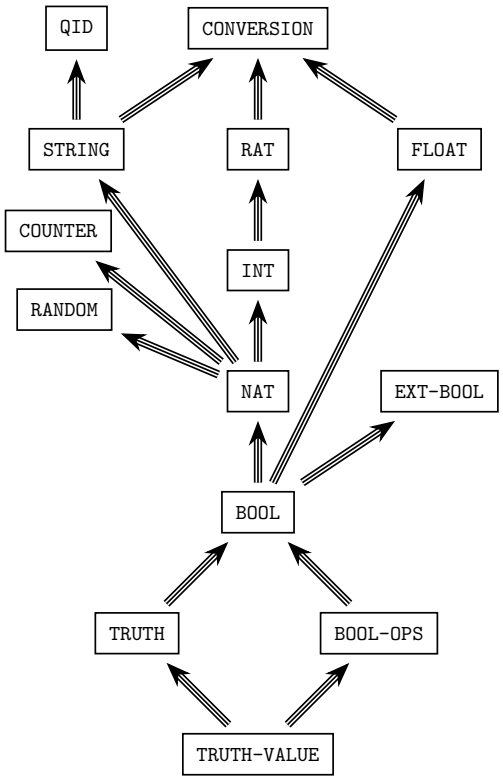


Figure 7.1: Importation (protecting) graph of predefined modules

7.1 Boolean values

There are five modules involving Boolean values, namely, `TRUTH-VALUE`, `TRUTH`, `BOOL-OPS`, `BOOL`, and `EXT-BOOL`. The most basic one is `TRUTH-VALUE`, which has the following definition.

```
fmod TRUTH-VALUE is
  sort Bool .
  op true : -> Bool [ctor special (...)] .
  op false : -> Bool [ctor special (...)] .
endfm
```

This module just declares the two Boolean values `true` and `false` as constants of sort `Bool`. The key thing to note is the `special` attribute associated with each of the operator declarations for these constants. In the case of Boolean values this is especially important, because certain basic constructs of the language such as conditions in a conditional equation, membership axiom, or rule, and also sort predicates associated with membership assertions evaluate to these built-in truth values.

The module `TRUTH` adds three important operators to `TRUTH-VALUE`.

```
fmod TRUTH is
  protecting TRUTH-VALUE .
  op if_then_else_fi : Bool Universal Universal -> Universal
       [poly (2 3 0) special (...)] .
```

```
  op _==_ : Universal Universal -> Bool
       [poly (1 2) prec 51 special (...)] .
  op _=/=_ : Universal Universal -> Bool
       [poly (1 2) prec 51 special (...)] .
endfm
```

The operators are, respectively, `if_then_else_fi`, and the built-in operators for equality and inequality predicates.[1] These operators are special in a number of ways. Firstly, they are, by default, automatically added to every module (see Section 3.9.3). Secondly, they are *polymorphic*, so that, for each module, they can be considered to be normal operators that are ad-hoc overloaded for each connected component in the module. This is done by means of the `polymorphic` (or `poly`) attribute, as discussed in Section 4.4.4, and the symbol `Universal`, that should not be considered a common sort, as explained at the end of this section. For example, in the declaration of the `if_then_else_fi` operator, the attribute `poly (2 3 0)` means that `if_then_else_fi` is polymorphic in its second and third arguments as well as in its result.

The `if_then_else_fi` operator first rewrites its first argument, the test. If the result is of sort `Bool`, the *then* or *else* argument is selected, according to whether the test evaluated to `true` or `false`, and rewritten. If the test result is not of sort `Bool` the *then* and *else* arguments are rewritten. For example, working in the `INT` module (see Section 7.4) we get the following reductions:

```
Maude> red in INT : if 4 - 2 == 2 then 0 else 1 fi .
result Zero: 0

Maude> red if 4 - 2 =/= 2 then 0 else 1 fi .
result NzNat: 1
```

The built-in Boolean predicates `_==_` and `_=/=_` have a straightforward operational meaning: given an expression `u == v` with `u` and `v` ground terms (i.e., terms without variables), then both `u` and `v` are simplified by the equations in the module (which are assumed to be Church-Rosser and terminating) to their canonical forms (perhaps modulo some axioms such as `assoc`, etc., see Section 4.4.1) and these canonical forms are compared for equality. If they are equal, the value of `u == v` is `true`; if they are different, it is `false`. The predicate `u =/= v` is just the negation of `u == v`.

Similar in spirit to the built-in operators for equality predicates, there are built-in operators for membership predicates: `_:: S` for each sort `S`. These do not need to be explicitly declared, because they are part of the extended signature associated with each module, as described in Section 3.9.3. The operational meaning for membership operators is analogous to that of the equality operators. Namely, given a term `u` and a sort `S` in its kind, the built-in predicate `u :: S` is evaluated by reducing `u` to its canonical form, computing its *least sort* (under the preregularity, Church-Rosser, and termination assumptions), and checking that it is smaller than or equal to `S`.

But what about the *mathematical* meaning of these built-in predicates? That is, do they really correspond to ordinary equations (and not to negations or other Boolean combinations of such equations)? The point is that these built-in and efficiently implemented equality and inequality predicates could in principle have been defined in a more cumbersome and inefficient way by the user. In fact, assuming that the equations and membership axioms in the user's module are Church-Rosser and terminating modulo the equational axioms in the operator attributes (see Section 4.4.1) and that the operators satisfy the preregularity requirement, the corresponding initial algebra is a *computable* algebraic data type, for which equality, inequality,

[1] The `prec` attribute in the last two operators assigns each of them an appropriate precedence value for parsing purposes (see Section 3.9).

and membership in a sort are also computable functions. Therefore, by a well-known theorem of Bergstra and Tucker [10], such predicates can themselves be equationally defined by Church-Rosser and terminating equations. It is of course very convenient, and much more efficient, to unburden the user from having to give those explicit equational definitions of the equality, inequality, and membership predicates by providing them in a built-in way.

Note also that, by the above meta-argument, the use of inequality predicates, negations of membership predicates, or any Boolean combination of such predicates in abbreviated Boolean conditions does not involve any real introduction of *negation* (or other Boolean connectives) in the underlying membership equational logic, which remains a Horn logic. What we are really doing is adding more Boolean-valued functions to the module, but such functions, although provided in a built-in way for convenience and efficiency, could have been equationally defined by the user without any use of negation.

The module `BOOL-OPS` imports `TRUTH-VALUE` and adds the usual conjunction, disjunction, exclusive or, negation, and implication operators.[2] These operators are defined entirely equationally.

```
fmod BOOL-OPS is
  protecting TRUTH-VALUE .
  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op not_ : Bool -> Bool [prec 53] .
  op _implies_ : Bool Bool -> Bool [gather (e E) prec 61] .
  vars A B C : Bool .
  eq true and A = A .
  eq false and A = false .
  eq A and A = A .
  eq false xor A = A .
  eq A xor A = false .
  eq A and (B xor C) = A and B xor A and C .
  eq not A = A xor true .
  eq A or B = A and B xor A xor B .
  eq A implies B = not(A xor A and B) .
endfm
```

Finally, the module `BOOL` puts together all the operators in `TRUTH` and in `BOOL-OPS`.

```
fmod BOOL is
  protecting BOOL-OPS .
  protecting TRUTH .
endfm
```

As noted above, the `BOOL` module is imported (in `including` mode) by default as a submodule of any other module defined by the user. This is accomplished by the command

```
set include BOOL on .
```

that appears in the standard library file `prelude.maude`. The `set include` command can mention any module we wish to import—in this case `BOOL`. However, this default importation can be disabled. For example, if the user wished to have the polymorphic equality, inequality and `if_then_else_fi` operators automatically added to modules, but wanted to exclude the usual Boolean connectives for the built-in truth values, this could be accomplished by writing

[2] See Section 3.9 for information on precedence values and gathering patterns.

```
set include BOOL off .
set include TRUTH on .
```

Similar commands are available for enabling and disabling implicit importation in `extending` and `protecting` modes (see Section 23.15). For example, the `BOOL` module can be protected by default instead of included by writing

```
set include BOOL off .
set protect BOOL on .
```

The module `EXT-BOOL` declares short-circuit versions of the conjunction and disjunction operators such as those found in LISP and other programming languages. Like the operators declared in `BOOL`, these operators are defined entirely equationally. The short-circuit behavior is the result of the strategy attributes declared for the operators as discussed in Section 4.4.7.

```
fmod EXT-BOOL is
  protecting BOOL .
  op _and-then_ : Bool Bool -> Bool
      [strat (1 0) gather (e E) prec 55] .
  op _or-else_ : Bool Bool -> Bool
      [strat (1 0) gather (e E) prec 59] .
  var B : [Bool] .
  eq true and-then B = B .
  eq false and-then B = false .
  eq true or-else B = true .
  eq false or-else B = B .
endfm
```

When the module `BOOL` is imported, the system automatically adds to the module an operator to test for membership, `_:: S`, for each sort `S`, as mentioned above (see also Section 3.9.3). Again, working in the `NUMBERS` module we have the following examples:

```
Maude> red in NUMBERS : sd(zero, zero) :: Zero .
result Bool: true

Maude> red sd(zero, zero) :: NzNat .
result Bool: false

Maude> red sd(zero, zero) :: Nat .
result Bool: true

Maude> red (zero nil) :: Zero .
result Bool: true
```

The term `sd(zero, zero)` reduces to `zero`, which has least sort `Zero` but also its supersort `Nat`. The term `zero nil` has also sort `Zero` because, using the equational axioms for the `assoc` and `id: nil` attributes, `zero nil` is the same as `zero`, which has sort `Zero`.

Note that these membership predicates are polymorphic on sorts, not on kinds. This is because to be syntactically well-formed the argument term must be of the right kind, namely the connected component containing the sort being tested. Thus a membership at the kind level is either trivially true or a syntactic error. Also, the presence of the system truth values is required for the predicates to be meaningful, so they are only added to modules that import the module `TRUTH-VALUE` (which is included by default, as part of `BOOL`, unless the user specifies otherwise).

Advisory. In fact, the symbol `Universal` does not denote a real sort: it is instead a place holder for parsing purposes that is given an interpretation by the `polymorphic` attribute (see

Section 4.4.4). The concrete effect of the interpretation of `Universal` is the instantiation in each connected component of the operators with one or more `Universal` arguments.

7.2 Natural numbers

The natural numbers module `NAT` provides a Peano-like specification of the natural numbers with an explicit successor function, while at the same time providing efficient built-in operators thanks to the `iter` theory (see Section 4.4.2) and an efficient binary representation of unbounded natural numbers arithmetic using the GNU GMP library.

The natural numbers sort hierarchy has top sort `Nat` and (disjoint) subsorts `Zero` and `NzNat`. The sort `Nat` is generated from the constant `0` (of sort `Zero`) and the successor operator `s_`.

```
fmod NAT is
  protecting BOOL .
  sorts Zero NzNat Nat .
  subsort Zero NzNat < Nat .
  *** constructors
  op 0 : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor iter special (...)] .
```

Having 0 and successor as constructors means that you can define functions on the natural numbers by matching into the successor notation; for example:

```
fmod FACTORIAL is
  protecting NAT .
  op _! : Nat -> NzNat .
  var N : Nat .
  eq 0 ! = 1 .
  eq (s N) ! = (s N) * N ! .
endfm
```

Try entering this module into Maude and then entering the commands

```
Maude> red 100 ! .
Maude> red 1000 ! .
```

(The results are omitted; the first has 158 digits and the second 2568 digits.)

Natural numbers can be input, and by default will be output, in normal decimal notation; however `42` is just syntactic sugar for `s_^42(0)`. The command `set print number on/off` controls whether or not decimal notation is used by the pretty printer. Thus executing the command `set print number off` will cause numbers to be printed using iteration notation.

Most of the usual arithmetic operators are provided in `NAT`. They are not defined algebraically but could be given an algebraic definition by the user if desired, for example for theorem proving purposes.

```
  *** ARITHMETIC OPERATIONS
  *** addition
  op _+_ : NzNat Nat -> NzNat [assoc comm prec 33 special (...)] .
  op _+_ : Nat Nat -> Nat [ditto] .
  *** symmetric difference
  op sd : Nat Nat -> Nat [comm special (...)] .
  *** multiplication
  op _*_ : NzNat NzNat -> NzNat [assoc comm prec 31 special (...)] .
  op _*_ : Nat Nat -> Nat [ditto] .
  *** quotient
```

```
op _quo_ : Nat NzNat -> Nat [prec 31 gather (E e) special (...)] .
*** remainder
op _rem_ : Nat NzNat -> Nat [prec 31 gather (E e) special (...)] .
*** exponential n^m = n * .... * n (m times)
op _^_ : Nat Nat -> Nat [prec 29 gather (E e) special (...)] .
op _^_ : NzNat Nat -> NzNat [ditto] .
*** exponential modulo modExp(n,m,p) = n^m mod p
op modExp : Nat Nat NzNat ~> Nat [special (...)] .
*** greatest common divisor
op gcd : NzNat NzNat -> NzNat [assoc comm special (...)] .
op gcd : Nat Nat -> Nat [ditto] .
*** least common multiple
op lcm : NzNat NzNat -> NzNat [assoc comm special (...)] .
op lcm : Nat Nat -> Nat [ditto] .
*** minimum
op min : NzNat NzNat -> NzNat [assoc comm special (...)] .
op min : Nat Nat -> Nat [ditto] .
*** maximum
op max : NzNat Nat -> NzNat [assoc comm special (...)] .
op max : Nat Nat -> Nat [ditto] .
```

The operators `_+_` and `_*_` compute the usual addition and multiplication operations and `_^_` is exponentiation.

The *symmetric difference* operator, `sd`, subtracts the smaller of its arguments from the larger. Thus, for example,

```
Maude> red in NAT : sd(4, 9) .
result NzNat: 5

Maude> red sd(9, 4) .
result NzNat: 5
```

The quotient and remainder operators, denoted `_quo_` and `_rem_`, satisfy the equation

$$((\texttt{i quo j}) * \texttt{j}) + (\texttt{i rem j}) = \texttt{i},$$

for natural numbers `i` and `j`. For example,

```
Maude> red in NAT : 11 quo 4 .
result NzNat: 2

Maude> red 11 rem 4 .
result NzNat: 3
```

The operator `modExp` computes modular exponentiation, with the third argument being the modulus. For example,

```
Maude> red in NAT : modExp(7, 1234, 2) .
result NzNat: 1

Maude> red modExp(8, 1234, 2) .
result Zero: 0
```

The operators `gcd`, `lcm`, `min`, and `max` compute the greatest common divisor, the least common multiple, the minimum and the maximum, respectively. Since these operators are associative and commutative, they can be used with any number (at least two) of arguments. For example,

```
Maude> red in NAT : gcd(6, 15, 21) .
result NzNat: 3

Maude> red lcm(6, 15, 21) .
result NzNat: 210

Maude> red min(6, 15, 21) .
result NzNat: 6

Maude> red max(6, 15, 21) .
result NzNat: 21

Maude> red gcd(0, 0) .
result Zero: 0
```

Operators that act on the binary representation of natural numbers interpreted as bit strings are:

- bitwise exclusive or (`_xor_`);
- bitwise and (`_&_`);
- bitwise or (`_|_`);
- rightshift—quotient by a power of 2 (`_>>_`); and
- leftshift—multiplication by a power of 2 (`_<<_`).

```
*** BITSTRING MANIPULATION
*** bitwise exclusive or
op _xor_ : Nat Nat -> Nat [assoc comm prec 55 special (...)] .
*** bitwise and
op _&_ : Nat Nat -> Nat [assoc comm prec 53 special (...)] .
*** bitwise or
op _|_ : NzNat Nat -> NzNat [assoc comm prec 57 special (...)] .
op _|_ : Nat Nat -> Nat [ditto] .
*** right shift -- quotient by power of 2
op _>>_ : Nat Nat -> Nat [prec 35 gather (E e) special (...)] .
*** left shift -- multiplication by power of 2
op _<<_ : Nat Nat -> Nat [prec 35 gather (E e) special (...)] .
```

Here are some examples using the bitwise operators.

```
Maude> red in NAT : 5 xor 7 .
result NzNat: 2

Maude> red 5 xor 2 .
result NzNat: 7

Maude> red 5 xor 5 .
result Zero: 0

Maude> red 5 & 7 .
result NzNat: 5
```

```
Maude> red 5 & 2 .
result Zero: 0

Maude> red 5 | 7 .
result NzNat: 7

Maude> red 5 | 2 .
result NzNat: 7

Maude> red 5 >> 2 .
result NzNat: 1

Maude> red 5 << 2 .
result NzNat: 20
```

The operators `_<_`, `_<=_`, `_>_`, and `_>=_` denote the usual operations for comparing numbers: less than, less than or equal, greater than, and greater than or equal, respectively. The operator `divides` returns true if and only if its second argument is a multiple of the first one.

```
  *** TESTS
  *** n less than m
  op _<_ : Nat Nat -> Bool [prec 37 special (...)] .
  *** n less than or equal to m
  op _<=_ : Nat Nat -> Bool [prec 37 special (...)] .
  *** n greater than m
  op _>_ : Nat Nat -> Bool [prec 37 special (...)] .
  *** n greater than or equal to m
  op _>=_ : Nat Nat -> Bool [prec 37 special (...)] .
  *** n divides m
  op _divides_ : NzNat Nat -> Bool [prec 51 special (...)] .
endfm
```

Note that, to avoid producing negative numbers, subtraction and bitwise not are not provided. The symmetric difference can be used in place of subtraction.

The operational semantics for most of the built-in operators is such that you only get built-in behavior when all the arguments are actually natural numbers. The exception is associative and commutative built-in operators which will compute as much as possible on natural number arguments and leave the remaining arguments unchanged; for example,

```
Maude> red in NAT : gcd(gcd(12, X:Nat), 15) .
result Nat: gcd(X:Nat, 3)
```

If the built-in operator does not disappear using the built-in semantics, then user equations are tried.

Advisory. It is easy to overload your machine's memory by generating huge natural numbers. There is a limit on exponentiation in that built-in behavior will not occur if the first argument is greater than 1 and the second argument is too large. Similarly, leftshift does not work if the first argument is greater than or equal to 1 and the second argument is too large. Currently "too large" means greater than 1000000 but this may change. Modular exponentiation has no such limits as its built-in semantics takes advantage of the fact that the result cannot be larger than the modulus. This is likely to be useful for cryptographic algorithms.

7.3 Random numbers and counters

The functional module `RANDOM` adds to `NAT` a pseudo-random number generator:

```
fmod RANDOM is
  protecting NAT .
  op random : Nat -> Nat [special (...)] .
endfm
```

The function `random` is the mapping from `Nat` into the range of natural numbers $[0, 2^{32} - 1]$ computed by successive calls to the Mersenne Twister Random Number Generator.[3] For example,

```
Maude> red in RANDOM : random(17) .
result NzNat: 1171049868
```

Although `random` is purely functional, it caches the state of the random number generator so that evaluating `random(0)` is always fast, as is evaluating `random(n+1)` if `random(n)` was the previous call to the operator `random`. In general, after generating `random(n)`, both `random(n)` and `random(n+1)` are computed efficiently because `random(n)` is a look up, while `random(n+k)` takes `k` steps of the twister or O(`k`) time.

By default the seed 0 is used, but a different seed, giving rise to a different function, may be specified by the command line option `-random-seed=`n, where n is a natural number in the range $[0, 2^{32} - 1]$. For example, if we invoke the Maude interpreter with the option `-random-seed=42` and run the previous example again we get

```
Maude> red in RANDOM : random(17) .
result NzNat: 613608295
```

The predefined system module `COUNTER` adds a "counter" that can be used to generate new names and new random numbers in Maude programs that do not want to explicitly maintain this state.

```
mod COUNTER is
  protecting NAT .
  op counter : -> [Nat] [special (...)] .
endm
```

For the `rewrite` and `frewrite` commands (see Sections 5.4 and 23.2), as well as the `erewrite` command (see later Section 9), the built-in constant `counter` has special rule rewriting semantics: each time it has the opportunity to do a rule rewrite, it rewrites to the next natural number, starting at 0. In this way the predefined system module `COUNTER` provides a built-in strategy for the application of the implicit nondeterministic rewrite rule

```
rl counter => N:Nat .
```

that rewrites the constant `counter` to a natural number. The built-in strategy applies this rule so that the natural number obtained after applying the rule is exactly the successor of the value obtained in the preceding rule application.

We can use the `COUNTER` module together with the predefined `RANDOM` module described above to sample various probability distributions. We illustrate the general idea with the following `SAMPLER` module, which can be used to sample a Bernoulli distribution corresponding to tossing a biased coin. This module also imports the predefined module `CONVERSION`, described later in Section 7.9, which includes conversion functions between different types of numbers.

[3] For information on the Mersenne Twister Random Number Generator, see `http://www-personal.engin.umich.edu/~wagnerr/MersenneTwister.html`.

```
mod SAMPLER is
  pr RANDOM .
  pr COUNTER .
  pr CONVERSION .
  op rand : -> [Float] .
  op sampleBernoulli : Float -> [Bool] .
  rl rand => float(random(counter) / 4294967295) .
  rl sampleBernoulli(P:Float) => rand < P:Float .
endm
```

The first rule rewrites the constant `rand` to a floating point number between 0 and 1 pseudo-randomly chosen according to the uniform distribution. This floating point number is obtained by converting the rational number `random(counter) / 4294967295` into a floating point number, where `4294967295`$= 2^{32} - 1$ is the maximum value that the `random` function can attain. We can then use the uniform sampling of a number between 0 and 1 to sample the tossing of a coin with a given bias `P:Float` between 0 and 1. This is accomplished by the second rewrite rule in `SAMPLER`.

Sampling capabilities defined in this style for different probability distributions can then be used to specify *probabilistic models* in Maude. We can give a flavor for how such models can be simulated in Maude by means of a simple battery-operated clock example. We may represent the state of such a clock as a term `clock(T,C)`, with `T` a natural number denoting the time, and `C` a positive real denoting the amount of battery charge. Each time the clock ticks, the time is increased by one unit, and the battery charge slightly decreases; however, the lower the battery charge, the greater the chance that the clock will stop, going into a state of the form `broken(T,C)`. We can model this system by means of the following Maude specification:

```
mod CLOCK is
  pr SAMPLER .
  sort Clock .
  op clock : Nat Float -> Clock [ctor] .
  op broken : Nat Float -> Clock [ctor] .
  var T : Nat .
  var C : Float .
  rl clock(T,C)
    => if sampleBernoulli(C / 1000.0)
       then clock(s(T), C - (C / 1000.0))
       else broken(T, C)
       fi .
endm
```

This rule models the fact that each time the clock is going to tick a coin is tossed; if the result is `true`, then the clock ticks normally, but if the result is `false`, then the clock breaks down. If the battery charge is high enough, the bias of the coin will be highly towards normal ticking, but as the battery charge decreases, the bias gradually decreases, so that a breakdown becomes increasingly likely.

One can use a module such as `CLOCK` above to perform *Monte Carlo simulations* of the probabilistic system we are interested in. Of course, we want different arguments for the random number generator to be used each time from the same initial state so that we obtain different behaviors. In Maude this can be easily achieved within the same Maude session by typing the command

```
set clear rules off .
```

which turns off the automatic clearing of rule state information, including counter values (see Section 23.2). This means that when we run several times the same computation, a different counter value will be initially used each time, therefore getting different behaviors in the expected Monte Carlo way. For example, we get the following simulations for the behavior of a clock until it breaks:

```
Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(40, 9.607702107358117e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(46, 9.5501998182355942e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(16, 9.8411944181564002e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(6, 9.9401498001499397e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(28, 9.7237474437709557e+2)
```

Since it is reasonable for a program to use multiple counters, the safe way to do this is to import renamed copies of `COUNTER`; for example

```
protecting COUNTER * (op counter to counter2) .
```

Counters are inactive with respect to search, model checking, and equational rewriting. Notice that there are potentially bad interactions with the debugger (see Section 20.1.3) since another `rewrite/frewrite/erewrite` executed in the debugger will lose the counter state of the interrupted `rewrite/frewrite/erewrite`.

7.4 Integer numbers

The module `INT` extends `NAT` with a unary minus `-_` on nonzero natural numbers to construct the negative integers. Integers can be input, and by default are output, in normal decimal notation; however, `-42` is just an alternative concrete syntax for `- 42`, which itself is just an alternative concrete syntax for `- s_^42(0)`.

```
fmod INT is
  protecting NAT .
  sorts NzInt Int .
  subsorts NzNat < NzInt Nat < Int .

  op -_ : NzNat -> NzInt [ctor special (...)] .
```

Unary minus is then extended to `Int` so that

```
  - - I:Int = I:Int
  - 0 = 0
```

The arithmetic operations of `NAT` are extended to integers. In addition, there are operators for subtraction, `_-_`, and absolute value, `abs`.

```
  *** ARITHMETIC OPERATIONS
  *** unary minus
  op -_ : NzInt -> NzInt [ditto] .
```

```
op -_ : Int -> Int [ditto] .
*** addition
op _+_ : Int Int -> Int [assoc comm prec 33 special (...)] .
*** subtraction
op _-_ : Int Int -> Int [prec 33 gather (E e) special (...)] .
*** multiplication
op _*_ : NzInt NzInt -> NzInt [assoc comm prec 31 special (...)] .
op _*_ : Int Int -> Int [ditto] .
*** quotient
op _quo_ : Int NzInt -> Int [prec 31 gather (E e) special (...)] .
*** remainder
op _rem_ : Int NzInt -> Int [prec 31 gather (E e) special (...)] .
*** exponentiation
op _^_ : Int Nat -> Int [prec 29 gather (E e) special (...)] .
op _^_ : NzInt Nat -> NzInt [ditto] .
*** absolute value
op abs : NzInt -> NzNat [special (...)] .
op abs : Int -> Nat [ditto] .
*** greatest common divisor
op gcd : NzInt NzInt -> NzNat [assoc comm special (...)] .
op gcd : Int Int -> Nat [ditto] .
*** least common multiple
op lcm : NzInt NzInt -> NzNat [assoc comm special (...)] .
op lcm : Int Int -> Nat [ditto] .
*** minimum
op min : NzInt NzInt -> NzInt [assoc comm special (...)] .
op min : Int Int -> Int [ditto] .
*** maximum
op max : NzInt NzInt -> NzInt [assoc comm special (...)] .
op max : Int Int -> Int [ditto] .
op max : NzNat Int -> NzNat [ditto] .
op max : Nat Int -> Nat [ditto] .
```

The operators `_quo_` and `_rem_` satisfy the same equation for integer arguments as for natural numbers. The sign of the quotient is the product of the signs of the arguments.

```
Maude> red in INT : -11 quo 4 .
result NzInt: -2

Maude> red 11 quo -4 .
result NzInt: -2

Maude> red -11 quo -4 .
result NzNat: 2

Maude> red 11 rem -4 .
result NzNat: 3

Maude> red -11 rem 4 .
result NzInt: -3

Maude> red -11 rem -4 .
result NzInt: -3
```

Bitwise operations on negative integers use the 2's complement representation and the operator `~_`, computing the bitwise not operation, is added.

```
  *** BITSTRING MANIPULATION (TWO'S COMPLEMENT)
  *** bitwise not
  op ~_ : Int -> Int [special (...)] .
  *** bitwise exclusive or
  op _xor_ : Int Int -> Int [assoc comm prec 55 special (...)] .
  *** bitwise and
  op _&_ : Nat Int -> Nat [assoc comm prec 53 special (...)] .
  op _&_ : Int Int -> Int [ditto] .
  *** bitwise or
  op _|_ : NzInt Int -> NzInt [assoc comm prec 57 special (...)] .
  op _|_ : Int Int -> Int [ditto] .
  *** rightshift
  op _>>_ : Int Nat -> Int [prec 35 gather (E e) special (...)] .
  *** leftshift
  op _<<_ : Int Nat -> Int [prec 35 gather (E e) special (...)] .
```

Tests on integers extend those on the natural numbers.

```
  *** TESTS
  *** less than
  op _<_ : Int Int -> Bool [prec 37 special (...)] .
  *** less than or equal
  op _<=_ : Int Int -> Bool [prec 37 special (...)] .
  *** greater than
  op _>_ : Int Int -> Bool [prec 37 special (...)] .
  *** greater than or equal
  op _>=_ : Int Int -> Bool [prec 37 special (...)] .

  op _divides_ : NzInt Int -> Bool [prec 51 special (...)] .
endfm
```

Let us show with an example how a predefined module can be reused to define new subsorts that refine the sort structure of the data type. In the following example, we introduce additional subsorts and overload the successor operator `s_` (originally coming from the module `NAT` imported in `protecting` mode into `INT`) in order to specify the sort of integers greater than three.

```
fmod INT-GT-3 is
  protecting INT .
  sorts One Two Three IntGt3 .
  subsorts One Two Three IntGt3 < NzNat .
  op s_ : Zero -> One [ctor ditto] .
  op s_ : One -> Two [ctor ditto] .
  op s_ : Two -> Three [ctor ditto] .
  op s_ : Three -> IntGt3 [ctor ditto] .
  op s_ : IntGt3 -> IntGt3 [ctor ditto] .
endfm
```

We can check the sort of a number by "reducing" the corresponding constant, as follows:

```
Maude> red -1 .
result NzInt: -1

Maude> red 0 .
result Zero: 0
```

```
Maude> red 1 .
result One: 1

Maude> red 2 .
result Two: 2

Maude> red 3 .
result Three: 3

Maude> red 4 .
result IntGt3: 4

Maude> red 12345678901234567890 .
result IntGt3: 12345678901234567890
```

In theory, the sort of integers greater than three could also be specified by means of membership axioms (see Sections 4.2 and 4.3). However, memberships are not guaranteed to work correctly with the number hierarchy, because of the special internal representation for iterated towers of `s_` symbols.

7.5 Machine integers

Versions of Maude prior to 2.0 supported machine integers in place of arbitrary size integers. Initially they were 32-bit in Maude 1.0 but were increased to 64-bit in Maude 1.0.5.

For certain applications, such as specifying programming languages that support machine integers as a built-in data type, it is convenient to have a predefined specification for machine integers. Fortunately, it is straightforward to efficiently emulate machine integers in terms of arbitrary size integers.

First we rename a copy of the regular integers, giving the sorts new names consistent with the new semantics and renaming those operators that either will not be defined on machine integers or else will have new semantics. Note that the operators `~_`, `_&_`, `_|_`, `_<_`, `_<=_`, `_>_`, and `_=>_` are not modified by the renaming.

```
fmod RENAMED-INT is
  protecting INT * (sort Zero to MachineZero,
                    sort NzNat to NzMachineNat,
                    sort Nat to MachineNat,
                    sort NzInt to NzMachineInt,
                    sort Int to MachineInt,
                    op s_ : Nat -> NzNat to $succ,
                    op sd : Nat Nat -> Nat to $sd,
                    op -_ : Int -> Int to $neg,
                    op _+_ : Int Int -> Int to $add,
                    op _-_ : Int Int -> Int to $sub,
                    op _*_ : NzInt NzInt -> NzInt to $mult,
                    op _quo_ : Int NzInt -> Int to $quo,
                    op _rem_ : Int NzInt -> Int to $rem,
                    op _^_ : Int Nat -> Int to $pow,
                    op abs : NzInt -> NzNat to $abs,
                    op gcd : NzInt Int -> NzNat to $gcd,
                    op lcm : NzInt NzInt -> NzNat to $lcm,
                    op min : NzInt NzInt -> NzInt to $min,
                    op max : NzInt NzInt -> NzInt to $max,
```

```
                  op _xor_ : Int Int -> Int to $xor,
                  op _>>_ : Int Nat -> Int to $shr,
                  op _<<_ : Int Nat -> Int to $shl,
                  op _divides_ : NzInt Int -> Bool to $divides) .
endfm
```

We then give a parameter theory that specifies the number of bits in a machine integer, which must be a power of 2, greater or equal to 2. Notice that this theory is based on the previous module, which is imported in `protecting` mode. Therefore, `$nrBits` is a *parameter constant* ranging over the `NzMachineNat` sort in the `RENAMED-INT` module, which is imported with an initial algebra semantics.

```
fth BIT-WIDTH is
  protecting RENAMED-INT .
  op $nrBits : -> NzMachineNat .

  var N : NzMachineNat .
  eq $divides(2, $nrBits) = true [nonexec] .
  ceq $divides(2, N) = true
    if $divides(N, $nrBits) /\ N > 1 [nonexec] .
endfth
```

Also provided are two predefined views that set the number of bits value `$nrBits` respectively to 32 and 64, the two most common sizes.

```
view 32-BIT from BIT-WIDTH to RENAMED-INT is
  op $nrBits to term 32 .
endv

view 64-BIT from BIT-WIDTH to RENAMED-INT is
  op $nrBits to term 64 .
endv
```

The module `MACHINE-INT` takes a bit width parameter and defines those operations that have a new semantics when applied to machine integers. In many cases this means applying the operation `$wrap` to the results to correctly simulate the wrap-around effect over an overflow on signed fixed bit width integers by, in effect, extending the sign bit infinitely to the left. In the case of `_^_` the meaning of the operation changes to exclusive or (from exponentiation on arbitrary size integers).

```
fmod MACHINE-INT{X :: BIT-WIDTH} is

  vars I J : MachineInt .
  var K : NzMachineInt .

  op $mask : -> NzMachineInt [memo] .
  eq $mask = $sub($nrBits, 1) .

  op $sign : -> NzMachineInt [memo] .
  eq $sign = $pow(2, $mask) .

  op maxMachineInt : -> NzMachineInt [memo] .
  eq maxMachineInt = $sub($sign, 1) .

  op minMachineInt : -> NzMachineInt [memo] .
  eq minMachineInt = $neg($sign) .
```

```
  op $wrap : MachineInt -> MachineInt .
  eq $wrap(I) = (I & maxMachineInt) | $neg(I & $sign) .

  op _+_ : MachineInt MachineInt -> MachineInt
      [assoc comm prec 33] .
  eq I + J = $wrap($add(I, J)) .

  op -_ : MachineInt -> MachineInt .
  eq - I = $wrap($neg(I)) .

  op _-_ : MachineInt MachineInt -> MachineInt
      [prec 33 gather (E e)] .
  eq I - J = $wrap($sub(I, J)) .

  op _*_ : MachineInt MachineInt -> MachineInt
      [assoc comm prec 31] .
  eq I * J = $wrap($mult(I, J)) .

  op _/_ : MachineInt NzMachineInt -> MachineInt
      [prec 31 gather (E e)] .
  eq I / K = $wrap($quo(I, K)) .

  op _%_ : MachineInt NzMachineInt -> MachineInt
      [prec 31 gather (E e)] .
  eq I % K = $rem(I, K) .

  op _^_ : MachineInt MachineInt -> MachineInt
      [prec 55 gather (E e)] .
  eq I ^ J = $xor(I, J) .

  op _>>_ : MachineInt MachineInt -> MachineInt
      [prec 35 gather (E e)] .
  eq I >> J = $shr(I, ($mask & J)) .

  op _<<_ : MachineInt MachineInt -> MachineInt
      [prec 35 gather (E e)] .
  eq I << J = $wrap($shl(I, ($mask & J))) .
endfm
```

Notice that using out of range integer constants may cause incorrect results.

We consider now the instantiation with the predefined view `32-BIT`, and show the wrap-around effect in several examples.

```
fmod MACHINE-INT-TEST is
  protecting MACHINE-INT{32-BIT} .
endfm
```

In the first examples, we can see the wrap-around from negative to positive and vice versa:

```
Maude> red -2147483648 - 1 .
result NzMachineNat: 2147483647

Maude> red 2147483647 + 1 .
result NzMachineInt: -2147483648
```

In the following product, the negative case does not wrap-around but the positive case does:

```
Maude> red -1073741824 * 2 .
result NzMachineInt: -2147483648

Maude> red 1073741824 * 2 .
result NzMachineInt: -2147483648
```

Division can only cause a wrap-around in this one case:

```
Maude> red -2147483648 / -1 .
result NzMachineInt: -2147483648
```

Remainder never wraps around:

```
Maude> red -2147483648 % -1 .
result MachineZero: 0
```

Finally, we see that the sign bit "falls off the left end" in a left shift:

```
Maude> red -2147483648 << 1 .
result MachineZero: 0
```

The parameterized `MACHINE-INT` module is an interesting example of Maude's support for what in type theory are called *dependent types* (see, for example, [100]). These are types like the power type `X[n]` or the `ARRAY{X,[n]}` type depending on a data parameter `n`, for example a natural number. We can view `MACHINE-INT` as the Maude analogue of a dependent type definition; however, note that the data parameter is not just any nonzero natural number, but must also satisfy *additional axioms*, specified in the `BIT-WIDTH` theory. For two other interesting examples of a Maude parameterized module defining the analogue of a dependent type, see the `POWER[n]` module in Section 21.3.1 (the exact analogue of the power type `X[n]`) and the `NAT/{N}` module of natural numbers modulo `N` in Section 22.7. Similarly, the `TUPLE[n]` module in Section 21.3.1 provides a form of dependent type that is not even available in some type theories with dependent types.

7.6 Rational numbers

The module `RAT` extends `INT` with a binary division operator `_/_` to construct the rationals from integers and nonzero naturals. Rationals can be input, and by default are output, in normal decimal notation; however `-5/42` is equivalent to `-5 / 42`, which is equivalent to `- 5 / 42`, which really denotes `- s_^5(0) / s_^42(0)`. The command

```
set print rat off .
```

switches off the special printing for `_/_` so that rational numbers will be printed with spaces around the foreslash sign. Note that `set print number off` also affects the printing of rational numbers, so with both number and rational pretty-printing switches turned off `-5/42` is printed using the final notation given above.

The numerator and denominator of a rational may contain common factors but these are removed by a single built-in rewrite whenever the rational is reduced (thus `_/_` is *not* a free constructor).

Notice that, in addition to the subsort `NzRat` of nonzero rational numbers, there is a subsort `PosRat` of positive rational numbers.

```
fmod RAT is
  protecting INT .
  sorts PosRat NzRat Rat .
  subsorts NzInt < NzRat Int < Rat .
  subsorts NzNat < PosRat < NzRat .

  op _/_ : NzInt NzNat -> NzRat
      [ctor prec 31 gather (E e) special (...)] .
  vars I J : NzInt .
  vars N M : NzNat .
  var  K : Int .
  var  Z : Nat .
  var  Q : NzRat .
  var  R : Rat .
```

The basic arithmetic operations on integers are extended to rational numbers as usual. The operator `_/_` is declared special for the case when the first argument is of sort `NzInt` to enhance performance. The remaining operators are defined in Maude by equations and may do some rewriting even when their arguments are not properly constructed rationals. Note that the choice of equations for defining operators on the rationals is motivated by performance: simpler equations are possible in many cases but they turn out to incur a big performance penalty.

```
  *** ARITHMETIC OPERATIONS
  op _/_ : NzNat NzNat -> PosRat [ctor ditto] .
  op _/_ : PosRat PosRat -> PosRat [ditto] .
  op _/_ : NzRat NzRat -> NzRat [ditto] .
  op _/_ : Rat NzRat -> Rat [ditto] .
  eq 0 / Q = 0 .
  eq I / - N = - I / N .
  eq (I / N) / (J / M) = (I * M) / (J * N) .
  eq (I / N) / J = I / (J * N) .
  eq I / (J / M) = (I * M) / J .

  op -_ : NzRat -> NzRat [ditto] .
  op -_ : Rat -> Rat [ditto] .
  eq - (I / N) = - I / N .

  op _+_ : PosRat PosRat -> PosRat [ditto] .
  op _+_ : PosRat Nat -> PosRat [ditto] .
  op _+_ : Rat Rat -> Rat [ditto] .
  eq I / N + J / M = (I * M + J * N) / (N * M) .
  eq I / N + K = (I + K * N) / N .

  op _-_ : Rat Rat -> Rat [ditto] .
  eq I / N - J / M = (I * M - J * N) / (N * M) .
  eq I / N - K = (I - K * N) / N .
  eq K - J / M = (K * M - J ) / M .

  op _*_ : PosRat PosRat -> PosRat [ditto] .
  op _*_ : NzRat NzRat -> NzRat [ditto] .
  op _*_ : Rat Rat -> Rat [ditto] .
  eq Q * 0 = 0 .
  eq (I / N) * (J / M) = (I * J) / (N * M).
```

```
eq (I / N) * K = (I * K) / N .

op _^_ : PosRat Nat -> PosRat [ditto] .
op _^_ : NzRat Nat -> NzRat [ditto] .
op _^_ : Rat Nat -> Rat [ditto] .
eq (I / N) ^ Z = (I ^ Z) / (N ^ Z) .

op abs : NzRat -> PosRat [ditto] .
op abs : Rat -> Rat [ditto] .
eq abs(I / N) = abs(I) / N .
```

The integer operations `quo`, `rem`, `gcd`, `lcm`, `min`, and `max` are also extended to the rational numbers. The operator `quo` gives the number of whole times a rational can be divided by another, `rem` gives the rational remainder. The operator `gcd` returns the largest rational that divides into each of its arguments a whole number of times, while `lcm` returns the smallest rational that is an integer multiple of its arguments.

```
op _quo_ : PosRat PosRat -> Nat [ditto] .
op _quo_ : Rat NzRat -> Int [ditto] .
eq (I / N) quo Q = I quo (N * Q) .
eq K quo (J / M) = (K * M) quo J .

op _rem_ : Rat NzRat -> Rat [ditto] .
eq (I / N) rem (J / M) = ((I * M) rem (J * N)) / (N * M) .
eq K rem (J / M) = ((K * M) rem J) / M .
eq (I / N) rem J = (I rem (J * N)) / N .

op gcd : NzRat Rat -> PosRat [ditto] .
op gcd : Rat Rat -> Rat [ditto] .
eq gcd(I / N, R) = gcd(I, N * R) / N .

op lcm : NzRat NzRat -> PosRat [ditto] .
op lcm : Rat Rat -> Rat [ditto] .
eq lcm(I / N, R) = lcm(I, N * R) / N .

op min : PosRat PosRat -> PosRat [ditto] .
op min : NzRat NzRat -> NzRat [ditto] .
op min : Rat Rat -> Rat [ditto] .
eq min(I / N, R) = min(I, N * R) / N .

op max : PosRat Rat -> PosRat [ditto] .
op max : NzRat NzRat -> NzRat [ditto] .
op max : Rat Rat -> Rat [ditto] .
eq max(I / N, R) = max(I, N * R) / N .
```

Some examples involving these operations are the following:

```
Maude> red in RAT : 1/2 quo 1/3 .
result NzNat: 1

Maude> red 1/2 rem 1/3 .
result PosRat: 1/6

Maude> red gcd(1/2, 1/3) .
result PosRat: 1/6
```

```
Maude> red lcm(1/2, 1/3) .
result NzNat: 1
```

Tests on integers are extended to rational numbers. The test `divides` returns true if a rational number divides another rational number a whole number of times.

```
*** tests
op _<_ : Rat Rat -> Bool [ditto] .
eq (I / N) < (J / M) = (I * M) < (J * N) .
eq (I / N) < K = I < (K * N) .
eq K < (J / M) = (K * M) < J .

op _<=_ : Rat Rat -> Bool [ditto] .
eq (I / N) <= (J / M) = (I * M) <= (J * N) .
eq (I / N) <= K = I <= (K * N) .
eq K <= (J / M) = (K * M) <= J .

op _>_ : Rat Rat -> Bool [ditto] .
eq (I / N) > (J / M) = (I * M) > (J * N) .
eq (I / N) > K = I > (K * N) .
eq K > (J / M) = (K * M) > J .

op _>=_ : Rat Rat -> Bool [ditto] .
eq (I / N) >= (J / M) = (I * M) >= (J * N) .
eq (I / N) >= K = I >= (K * N) .
eq K >= (J / M) = (K * M) >= J .

op _divides_ : NzRat Rat -> Bool [ditto] .
eq (I / N) divides K = I divides N * K .
eq Q divides (J / M) = Q * M divides J .
```

There are four new operators: `trunc`, `frac`, `floor`, and `ceiling`. The operator `floor` converts a rational number to an integer by rounding down to the nearest integer, `ceiling` rounds up, and `trunc` rounds towards 0. The operator `frac` gives the fraction part of its argument and this always has the same sign as its argument.

```
*** ROUNDING
op trunc : PosRat -> Nat .
op trunc : Rat -> Int .
eq trunc(K) = K .
eq trunc(I / N) = I quo N .

op frac : Rat -> Rat .
eq frac(K) = 0 .
eq frac(I / N) = (I rem N) / N .

op floor : PosRat -> Nat .
op floor : Rat -> Int .
eq floor(K) = K .
eq floor(N / M) = N quo M .
eq floor(- N / M) = - ceiling(N / M) .

op ceiling : PosRat -> NzNat .
op ceiling : Rat -> Int .
eq ceiling(K) = K .
eq ceiling(N / M) = ((N + M) - 1) quo M .
```

```
  eq ceiling(- N / M) = - floor(N / M) .
endfm
```

Here are some examples of reductions involving the rounding operators:

```
Maude> red in RAT : trunc(9/7) .
result NzNat: 1

Maude> red floor(9/7) .
result NzNat: 1

Maude> red ceiling(9/7) .
result NzNat: 2

Maude> red frac(9/7) .
result PosRat: 2/7

Maude> red trunc(-9/7) .
result NzInt: -1

Maude> red floor(-9/7) .
result NzInt: -2

Maude> red ceiling(-9/7) .
result NzInt: -1

Maude> red frac(-9/7) .
result NzRat: -2/7
```

7.7 Floating-point numbers

The module `FLOAT` declares sorts and operators for manipulating floating-point numbers, which are implemented using double precision floating-point arithmetic of the underlying hardware platform, conforming to the IEEE-754 standard when supported by the hardware platform. Floating-point numbers are treated as a large set of constants, that is, a floating-point number has no algebraic structure (this is the reason for the special operator declaration `<Floats>`, as explained in the introduction of this chapter).

The sort `FiniteFloat` consists of the floating-point numbers that have a 64-bit representation. Finite floating-point numbers can be input, and by default are output, in scientific notation; they can also be input using decimal point notation. Thus `100.0` is equivalent to `1.0e+2`. The constants `Infinity` and `-Infinity` represent floating-point numbers that are outside the 64-bit representable range. Thus `Infinity` and `-Infinity` are of sort `Float` but not of sort `FiniteFloat`. Note that there are some surprises when using decimal notation to input floating-point numbers. For example, in the `FLOAT` module we have the reduction

```
Maude> red in FLOAT : 1.1 .
result FiniteFloat: 1.1000000000000001
```

This is because floating-point numbers are represented internally using a binary expansion rather than a decimal expansion and 1.1 does not have a finite length binary expansion.

```
fmod FLOAT is
  protecting BOOL .
  sorts FiniteFloat Float .
  subsort FiniteFloat < Float .
  op <Floats> : -> FiniteFloat [special (...)] .
  op <Floats> : -> Float [ditto] .
```

The arithmetic operators `-_`, `_-_`, `_+_`, `_*_`, `_/_`, `_^_`, and `abs` have the usual interpretation, as in the module `INT`. Note that `1.2 / 0.0` is just an expression of kind `[Float]` and reducing it does not cause your system to crash!

```
  *** ARITHMETIC OPERATIONS
  op -_ : Float -> Float [prec 15 special (...)] .
  op -_ : FiniteFloat -> FiniteFloat [ditto] .

  op _+_ : Float Float -> Float
      [prec 33 gather (E e) special (...)] .
  op _-_ : Float Float -> Float
      [prec 33 gather (E e) special (...)] .
  op _*_ : Float Float -> Float
      [prec 31 gather (E e) special (...)] .
  op _/_ : Float Float ~> Float
      [prec 31 gather (E e) special (...)] .
  op _^_ : Float Float ~> Float
      [prec 29 gather (E e) special (...)] .

  op abs : Float -> Float [special (...)] .
  op abs : FiniteFloat -> FiniteFloat [ditto] .
```

The operator `_rem_` computes the remainder of a division, `floor` rounds down to the nearest integer, `ceiling` rounds up, and `sqrt` computes the square root.

```
  op _rem_ : Float Float ~> Float
      [prec 31 gather (E e) special (...)] .
  op floor : Float -> Float [special (...)] .
  op ceiling : Float -> Float [special (...)] .
  op sqrt : Float ~> Float [special (...)] .
```

For terms `f1` and `f2` of sort `FiniteFloat`, `f1 rem f2` computes the remainder of dividing `f1` by `f2`. Specifically, `f1 rem f2` is equal to `f1 - n * f2`, where `n` is `f1 / f2` rounded towards zero to the nearest integer. For example,

```
Maude> red in FLOAT : 5.0 rem 2.0 .
result FiniteFloat: 1.0

Maude> red -5.0 rem 2.0 .
result FiniteFloat: -1.0

Maude> red 5.0 rem 2.5 .
result FiniteFloat: 0.0
```

Some examples of reductions using the `floor` and `ceiling` operations are the following:

```
Maude> red in FLOAT : ceiling(2.5) .
result FiniteFloat: 3.0
```

```
Maude> red floor(2.5) .
result FiniteFloat: 2.0

Maude> red ceiling(- 2.5) .
result FiniteFloat: -2.0

Maude> red floor(- 2.5) .
result FiniteFloat: -3.0

Maude> red ceiling(Infinity) .
result Float: Infinity

Maude> red floor(-Infinity) .
result Float: -Infinity
```

The operators `max` and `min` for computing the maximum and the minimum, respectively, work as expected,

```
op min : Float Float -> Float [special (...)] .
op max : Float Float -> Float [special (...)] .
```

as we can see in the following examples:

```
Maude> red in FLOAT : min(2.0, -2.0) .
result FiniteFloat: -2.0

Maude> red max(2.0, -2.0) .
result FiniteFloat: 2.0

Maude> red max(2.0, Infinity) .
result Float: Infinity

Maude> red in FLOAT : min(Infinity, -Infinity) .
result Float: -Infinity
```

The operators `exp` and `log` compute the natural exponent and logarithm, respectively.

```
*** TRANSCENDENTAL OPERATIONS
op exp : Float -> Float [special (...)] .
op log : Float ~> Float [special (...)] .
```

Here are some examples:

```
Maude> red in FLOAT : exp(1.0) .
result FiniteFloat: 2.7182818284590451

Maude> red log(exp(1.0)) .
result FiniteFloat: 1.0

Maude> red log(0.0) .
result Float: -Infinity
```

The constant `pi` approximates the value of π. The number of digits is chosen to be the largest that can accurately be represented as a floating-point number. The trigonometric operators `sin`, `cos`, and `tan` expect arguments in radians. The operators `asin`, `acos`, `atan` are the corresponding inverses.

```
*** TRIGONOMETRIC OPERATIONS
op sin : Float -> Float [special (...)] .
op cos : Float -> Float [special (...)] .
op tan : Float -> Float [special (...)] .
op asin : Float ~> Float [special (...)] .
op acos : Float ~> Float [special (...)] .
op atan : Float -> Float [special (...)] .
op atan : Float Float -> Float [special (...)] .

op pi : -> FiniteFloat .
eq pi = 3.1415926535897931 .
```

Here are some examples of reductions of trigonometric expressions.

```
Maude> red in FLOAT : sin(0.0) .
result FiniteFloat: 0.0

Maude> red sin(pi) .
result FiniteFloat: 1.2246467991473532e-16

Maude> red cos(pi) .
result FiniteFloat: -1.0

Maude> red acos(cos(pi)) .
result FiniteFloat: 3.1415926535897931

Maude> red tan(pi) .
result FiniteFloat: -1.2246467991473532e-16

Maude> red sin(pi / 2.0) .
result FiniteFloat: 1.0

Maude> red cos(pi / 2.0) .
result FiniteFloat: 6.123233995736766e-17

Maude> red tan(pi / 2.0) .
result FiniteFloat: 1.633123935319537e+16

Maude> red atan(tan(pi / 2.0)) .
result FiniteFloat: 1.5707963267948966

Maude> red pi / 2.0 .
result FiniteFloat: 1.5707963267948966
```

Using the binary form of the arc tangent operator, `atan(f1, f2)`, is similar to computing `atan(f1 / f2)`, except that the signs of both arguments are used to control the quadrant of the result.

```
Maude> red in FLOAT : atan(tan(pi / 3.0)) .
result FiniteFloat: 1.0471975511965976

Maude> red atan(tan(pi / 3.0), 1.0) .
result FiniteFloat: 1.0471975511965976
```

```
Maude> red atan(tan(pi / 3.0), -1.0) .
result FiniteFloat: 2.0943951023931957

Maude> red atan(- tan(pi / 3.0), -1.0) .
result FiniteFloat: -2.0943951023931957

Maude> red atan(- tan(pi / 3.0), 1.0) .
result FiniteFloat: -1.0471975511965976
```

Numerical comparisons have the usual meaning on floating-point numbers.

```
  *** TESTS
  op _<_ : Float Float -> Bool [prec 51 special (...)] .
  op _<=_ : Float Float -> Bool [prec 51 special (...)] .
  op _>_ : Float Float -> Bool [prec 51 special (...)] .
  op _>=_ : Float Float -> Bool [prec 51 special (...)] .

  *** approximate equality
  op _=[_]_ : Float FiniteFloat Float -> Bool [prec 51] .
  vars X Y : Float .
  var  Z : FiniteFloat .
  eq X =[Z] Y = abs(X - Y) < Z .
endfm
```

The operator `_=[_]_` tests for approximate equality, where the second argument bounds the allowed error. For example:

```
Maude> red in FLOAT : 1.111111111 =[1.0e-9] 1.111111112 .
result Bool: true

Maude> red 1.111111111 =[1.0e-10] 1.111111112 .
result Bool: false
```

7.8 Strings

The module `STRING` declares sorts and operators for manipulating strings of characters. Strings of length one form a subsort `Char` of `String`. Operations on strings are based on the rope data structure [11], which has been optimized for functional programming, where copying with modification is supported efficiently, whereas arbitrary in-place updates are not.

Strings are input and output using the usual convention of enclosing the string characters in a pair of matching quotes `"..."`. When a string is parsed, it is interpreted using a subset of ANSI C backslash escape conventions [87, Section A2.5.2].

To define the results of searching a string for an occurrence of another substring the sort `FindResult` is introduced. This sort consists of the natural numbers, returned as the index in the string where a found substring begins (string indexing begins with 0), and a special constant `notFound`, returned if no occurrence is found.

```
fmod STRING is
  protecting NAT .
  sorts String Char FindResult .
  subsort Char < String .
  subsort Nat < FindResult .
  op <Strings> : -> Char [special (...)] .
  op <Strings> : -> String [ditto] .
  op notFound : -> FindResult [ctor] .
```

The operators `ascii` and `char` convert between characters and ASCII codes.

```
*** conversion between ascii code and character
op ascii : Char -> Nat [special (...)] .
op char : Nat ~> Char [special (...)] .
```

For a natural number `n` less than 256 and a character `c`, we have `ascii(char(n)) = n` and `char(ascii(c)) = c`. For a natural number `n` greater than 255, `char(n)` is an error term of kind `[String]`. For example,

```
Maude> red in STRING : ascii("#") .
result NzNat: 35

Maude> red char(35) .
result Char: "#"

Maude> red ascii("a") .
result NzNat: 97

Maude> red char(97) .
result Char: "a"

Maude> red char(255) .
result Char: "\377"
```

On strings, `_+_` denotes the concatenation operation, with identity the empty string, `""`. String length is computed by the `length` operator.

```
*** string concatenation
op _+_ : String String -> String
    [prec 33 gather (E e) special (...)] .

*** string length
op length : String -> Nat [special (...)] .
```

Here are some examples.

```
Maude> red in STRING : "abc" + "def" .
result String: "abcdef"

Maude> red "ab" + "cd" + "ef" .
result String: "abcdef"

Maude> red "abc" + "" .
result String: "abc"

Maude> red length("abcdef") .
result NzNat: 6

Maude> red length("") .
result Zero: 0
```

The operators `substr`, `find`, and `rfind` deal with finding and extracting substrings. Remember that string indexing begins with 0.

```
*** substring
*** second argument is starting position, third is length
op substr : String Nat Nat -> String [special (...)] .

*** starting position of substring (second argument)
*** least one >= third argument (find)
*** greatest one <= third argument (rfind)
op find : String String Nat -> FindResult [special (...)] .
op rfind : String String Nat -> FindResult [special (...)] .
```

The expression `substr(S:String, Start:Nat, Len:Nat)` returns the substring of `S:String` of length `Len:Nat` beginning at position `Start:Nat`. If the value of the term `Start:Nat + Len:Nat` is greater than `length(S:String)` then the returned substring is the tail of `S:String` starting from position `Start:Nat`. This will be empty if the starting position is past the end of the string.

```
Maude> red in STRING : substr("abc", 0, 2) .
result String: "ab"

Maude> red substr("abc", 1, 2) .
result String: "bc"

Maude> red substr("abc", 1, 3) .
result String: "bc"

Maude> red substr("abc", 3, 2) .
result String: ""
```

`find` searches for the first match from the beginning of the string, while `rfind` searches from the end of the string backwards.

`find(S:String, Pat:String, Start:Nat)` returns the least index of an occurrence of `Pat:String` in `S:String` that is greater than or equal to `Start:Nat`. If no such index exists the constant `notFound` is returned.

`rfind(S:String, Pat:String, Start:Nat)` returns the greatest index of an occurrence of `Pat:String` in `S:String` that is less than or equal to `Start:Nat`. If no such index exists the constant `notFound` is returned.

```
Maude> red in STRING : find("abc", "b", 0) .
result NzNat: 1

Maude> red find("abc", "b", 1) .
result NzNat: 1

Maude> red find("abc", "b", 2) .
result FindResult: notFound

Maude> red find("abc", "d", 2) .
result FindResult: notFound

Maude> red rfind("abc", "b", 2) .
result NzNat: 1

Maude> red rfind("abc", "b", 1) .
result NzNat: 1
```

```
Maude> red rfind("abc", "b", 0) .
result FindResult: notFound

Maude> red rfind("abc", "d", 2) .
result FindResult: notFound
```

Some properties relating `substr`, `find`, and `rfind` are the following, where S and P are variables of sort `String`, and I, J, and K are variables of sort `Nat` such that `length(`S`)` $= K$ and `length(`P`)` $= J$.

$I \leq$ `find(`S, P, I`)` $\leq K$`-`J

`0` $\leq$ `rfind(`S, P, I`)` $\leq$ `min(`I, K`-`J`)`

`find(`$S, S,$ `0)` $=$ `0` $=$ `rfind(`S, S, I`)`

`find(`$S,$ `"", `I`)` $=$ *if* $I \leq K$ *then* I *else* `notFound`

`rfind(`$S,$ `"", `I`)` $=$ *if* $I \geq K$ *then* K *else* I

`find(`S, P, I`)` $\neq$ `notFound`
$\implies$ `substr(`$S,$ `0,` `find(`S, P, I`)) + `P ` + substr(`$S,$ `find(`S, P, I`)+`J, K`)` $= S$

`rfind(`S, P, I`)` $\neq$ `notFound`
$\implies$ `substr(`$S,$ `0,` `rfind(`S, P, I`)) + `P ` + substr(`$S,$ `rfind(`S, P, I`)+`J, K`)` $= S$

The operators `_<_`, `_<=_`, `_>_`, and `_>=_` denote string comparison operations using the lexicographic order, where characters are compared going through their ASCII codes.

```
  *** lexicographic string comparison
  op _<_ : String String -> Bool [prec 37 special (...)] .
  op _<=_ : String String -> Bool [prec 37 special (...)] .

  op _>_ : String String -> Bool [prec 37 special (...)] .
  op _>=_ : String String -> Bool [prec 37 special (...)] .
endfm
```

Here are some examples.

```
Maude> red in STRING : "abc" < "abd" .
result Bool: true

Maude> red "abc" < "abb" .
result Bool: false

Maude> red "abc" < "abcd" .
result Bool: true
```

7.9 String and number conversions

The module `CONVERSION` consolidates all the conversion functions between the three major built-in data types: `Nat/Int/Rat`, `Float`, and `String`.

```
fmod CONVERSION is
  protecting RAT .
  protecting FLOAT .
  protecting STRING .
```

```
*** number type conversions
op float : Rat -> Float [special (...)] .
op rat : FiniteFloat -> Rat [special (...)] .
```

The operation `float` computes the floating-point number nearest to a given rational number. If the value of the rational number falls outside the range representable by IEEE-754 double precision finite floating-point numbers, `Infinity` or `-Infinity` is returned as appropriate. This is in accord with the convention that `Infinity` and `-Infinity` are used to handle out-of-range situations in the floating-point world.

The operator `rat` converts finite floating-point numbers to rational numbers exactly (since every IEEE-754 finite floating-point number is a rational number). Of course, if the result happens to be a natural number or an integer, that is what you get. `rat(Infinity)` and `rat(-Infinity)` do not reduce, since they have no reasonable representation in the world of rational numbers. It is intended that the equation

```
float(rat(F:FiniteFloat)) = F:FiniteFloat
```

is satisfied, although this holds only if the third party library (GNU GMP) being used in the implementation meets its related requirements.

```
*** string <-> number conversions
op string : Rat NzNat ~> String [special (...)] .
op rat : String NzNat ~> Rat [special (...)] .
op string : Float -> String [special (...)] .
op float : String ~> Float [special (...)] .
```

The operator `string` converts a rational number to a string using a given base, which must lie in the range 2..36. Rational numbers that are really natural numbers or integers are converted to string representations of natural numbers or integers, so we have for example

```
Maude> red in CONVERSION : string(-1, 10) .
result String: "-1"
```

The operator `rat` converts a string to a rational number using a given base, which must lie in the range 2..36. Of course, if the result happens to be a natural number or an integer, that is what you get. Currently the function is very strict about which strings are converted: the string must be something that the Maude parser would recognize as a natural number, an integer or a rational number. This could be changed to a more generous interpretation in the future.

The operators `string` and `float` for conversion between floating-point numbers and strings satisfy the equation

```
float(string(F:Float)) = F:Float
```

A new sort, `DecFloat`, is introduced to provide the means for arbitrary formatting of floating-point numbers.

```
  sort DecFloat .
  op <_,_,_> : Int String Int -> DecFloat [ctor] .
  op decFloat : Float Nat -> DecFloat [special (...)] .
endfm
```

A `DecFloat` consists of a sign (1, 0 or -1), a string of digits, and a decimal point position (0 is just in front of first digit, $-n$ is n positions to the left, and $+n$ is n positions to the right). Thus, `< -1, "123", 11 >` represents `-1.23e10`. `decFloat(F, N)` converts `F` to a `DecFloat`, rounding to `N` significant digits using the IEEE-754 "round to nearest" rule with

trailing zeros if needed. If `N` is 0, an *exact* `DecFloat` representation of `F` is produced—this may require hundreds of digits. For any natural number `N`, `decFloat(Infinity, N)` reduces to `< 1, "Infinity", 0 >`. Here are some examples.

```
Maude> red in CONVERSION : decFloat(Infinity, 9) .
result DecFloat: < 1,"Infinity",0 >

Maude> red decFloat(-Infinity, 9) .
result DecFloat: < -1,"Infinity",0 >

Maude> red decFloat(123.0, 5) .
result DecFloat: < 1,"12300",3 >

Maude> red decFloat(-123.0, 5) .
result DecFloat: < -1,"12300",3 >

Maude> red decFloat(.123, 5) .
result DecFloat: < 1,"12300",0 >

Maude> red decFloat(.00123, 5) .
result DecFloat: < 1,"12300",-2 >

Maude> red decFloat(0.0, 5) .
result DecFloat: < 0,"00000",0 >
```

Advisory. Counterintuitive results are possible when converting from the approximate world of floating-point numbers to the exact world of rational numbers. For example,

```
Maude> red in CONVERSION : rat(1.1) .
result PosRat: 2476979795053773/2251799813685248
```

This is because, as mentioned above, 1.1 cannot be represented exactly as a floating-point number, and the nearest floating-point number is

```
1.1000000000000000888178419700125232338905334472656 25
```

which is the above rational number. (Note that Maude prints the number 1.1 as 1.1000000000000001, using 17 significant digits. The above representation is obtained by reducing `decFloat(1.1, 52)`.)

7.10 Quoted identifiers

The module `QID` is a wrapper for strings in order to provide a Maude representation for tokens of Maude syntax. Quoted identifiers are input and output by preceding a Maude identifier[4] with a (fore) quote sign. Thus `'abc` is a quoted identifier whose underlying string is `"abc"`. A quoted identifier is also an identifier, as are strings. Thus `''abc` and `'"abc"` are both quoted identifiers.

```
fmod QID is
  protecting STRING .
  sort Qid .
  op <Qids> : -> Qid [special (...)] .
```

[4]The syntax of Maude identifiers is discussed in Section 3.1.

```
  *** qid <-> string conversions
  op string : Qid -> String [special (...)] .
  op qid : String ~> Qid [special (...)] .
endfm
```

The operators `qid` and `string` do the wrapping and unwrapping. `string` is injective, since every quoted identifier has a unique corresponding string.

```
Maude> red in QID : string('abc) .
result String: "abc"

Maude> red qid("abc") .
result Qid: 'abc

Maude> red string('a\b) .
result String: "a\\b"

Maude> red qid("a\\b") .
result Qid: 'a\b

Maude> red string('a`[b) .
result String: "a`[b"

Maude> red qid("a[b") .
result Qid: 'a`[b
```

The operator `qid` is only injective on strings without white space, control characters, and certain other characters which are converted to backquote. Thus the equation `qid(string(q)) = q` holds for quoted identifiers `q`.

```
Maude> red in QID : qid("a b c") .
result Qid: 'a`b`c

Maude> red string('a`b`c) .
result String: "a`b`c"

Maude> red qid("a\t b") .
result Qid: 'a`b

Maude> red string('a`b) .
result String: "a`b"
```

An example of a string that cannot be converted to a quoted identifier is `"a\"b"` since identifiers are not allowed to have unpaired double quotes. Thus `qid("a\"b")` has kind `[Qid]` but does not reduce to something of sort `Qid`.

7.11 Conversions between strings and lists of quoted identifiers

The module `LEXICAL` provides support for converting between strings and lists of quoted identifiers using Maude's lexical conventions.

```
fmod LEXICAL is
  protecting QID-LIST .
  op printTokens : QidList -> String [special (...)] .
  op tokenize : String -> QidList [special (...)] .
endfm
```

`printTokens()` converts each quoted identifier to a string and concatenates with spaces where appropriate.

The conversion is slightly different from the `string` function in the `QID` module (see Section 7.10). For simple quoted identifers the conversions are the same:

```
printTokens('abc) = string('abc) = "abc"
```

However, there are a number of character sequences that are not legitimate identifiers, and hence do not have quoted identifiers as such, but are useful and so are instead represented as quoted identifiers using the `metaPrettyPrint()/LOOP-MODE` conventions:

- For the seven special tokens `(`, `)`, `[`, `]`, `{`, `}`, and `,` the conversions of the associated quoted identifiers differ, with `printTokens()` returning a one-character string while `string()` includes the backquote:

  ```
  printTokens('`() = "(" whereas string('`()) = "`("
  ```

- For quoted identifiers of the form `'\x` where `x` is a single character, there are several possibilities. If `x` is `n`, `t`, or `\`, then `printTokens()` understands the C escape convention:

  ```
  printTokens('\n) = "\n" whereas string('\n) = "\\n"
  ```

- The special case of `'\s` is used to have a quoted identifier that represents a single space:

  ```
  printTokens('\s) = " " whereas string('\s) = "\\s"
  ```

- The 21 special values of x:

  `!`, `?`, `u`, `f`, `x`, `h`, `o`, `p`, `r`, `g`, `y`, `m`, `c`, `w`, `P`, `R`, `G`, `Y`, `M`, `X`, and `W`

  are used to represent ANSI control sequences. For example:

  ```
  printTokens('\r) = "\033[31m" whereas string('\r) = "\\r"
  ```

  If `x` has any other value, `printTokens()` behaves the same as `string()` to produce a two-character string (though of course when printing this string as an identifier, it is surrounded by double quotes and the itself is escaped):

  ```
  printTokens('\q) = string('\q) = "\\q"
  ```

  `printTokens()` also works on lists of quoted identifers. The `nil` list is mapped to the empty string:

  ```
  printTokens(nil) = ""
  ```

  Lists with more than one quoted identifier map to the concatenation of strings produced for each quoted identifier, with a single space inserted between such strings with two exceptions:

  - No space is inserted before `(`, `)`, `[`, `]`, `{`, `}`, `,`.
  - No space is inserted after `(`, `[`, `{`.

For example:

```
reduce in LEXICAL : printTokens('f ''( 'a '', 'b '') '+ 'c) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "f(a, b) + c"
```

- Furthermore the quoted-identifiers forms of the seven special symbols that include a backslash are not recognized as special:

```
printTokens('\'() = string('\'() = "\\'("
```

`tokenize()` converts a string into a list of quoted identifers. Characters that cannot be part of a token terminate any partial token and are otherwise ignored.

```
reduce in LEXICAL : tokenize("f(a, b) + c") .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result NeQidList: 'f ''( 'a '', 'b '') '+ 'c

reduce in LEXICAL : tokenize("      ") .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result QidList: nil

reduce in LEXICAL : tokenize("") .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result QidList: nil

reduce in LEXICAL : tokenize("\"string identifier\"") .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Qid: '"string identifier"
```

In particular, an unclosed double quote will terminate the current token but otherwise be ignored:

```
reduce in LEXICAL : tokenize("foo\"bar") .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result NeQidList: 'foo 'bar
```

A closed double quote is a legitimate continuation of the current token:

```
reduce in LEXICAL : tokenize("foo\"bar\"") .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Qid: 'foo"bar"
```

Backslash-newline pairs inside double quotes are edited out:

```
reduce in LEXICAL : tokenize("\"foo\\\nbar\"") .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Qid: '"foobar"
```

7.12 Basic theories and standard views

The library of predefined modules provided by Maude in the `prelude.maude` file includes some well-known parameterized data types that will be described in the following sections. Here we will introduce the standard theories that provide the requirements for those parameterized modules.

7.12.1 TRIV

As already described in Section 6.3.1, the simplest non-empty theory is called TRIV and consists of a single sort. A model of this theory is just a set of any cardinality (finite or infinite). The intuition behind this simple theory is that the minimum requirement possible on a parameterized data type construction is having a data type as a set of basic elements to build more data on top of it. For example, in the LIST{X :: TRIV} parameterized data type construction we need a data type (set) of basic elements satisfying TRIV to then build lists of such elements.

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

The file prelude.maude includes many views out of TRIV that select the main sort of the built-in modules that we have already described in the previous sections. All these views are named in the same way: by the sort they select; for example, the standard view from TRIV into RAT selecting the sort Rat is also named Rat.

```
view Bool from TRIV to BOOL is
  sort Elt to Bool .
endv

view Nat from TRIV to NAT is
  sort Elt to Nat .
endv

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

view Rat from TRIV to RAT is
  sort Elt to Rat .
endv

view Float from TRIV to FLOAT is
  sort Elt to Float .
endv
```

et

```
view String from TRIV to STRING is
  sort Elt to String .
endv

view Qid from TRIV to QID is
  sort Elt to Qid .
endv
```

7.12.2 DEFAULT

The theory DEFAULT is slightly more complex than TRIV, in that in addition to a sort it also requires that there be a distinguished "default" element in such a sort. Notice that DEFAULT imports TRIV in the following presentation:

```
fth DEFAULT is
  including TRIV .
  op 0 : -> Elt .
endfth
```

The inclusion of the theory `TRIV` into the theory `DEFAULT` is made explicit by the following view, whose name coincides with the name of the target theory.

```
view DEFAULT from TRIV to DEFAULT is
endv
```

The Maude library also includes several views that map from `DEFAULT` to the various built-in data type modules by selecting the main sort and a distinguished element in it. In the case of the number sorts, this element is the zero, while for strings it is the empty string and for quoted identifiers is just the quote. Notice that operator mappings that are the identity (i.e., of the form `op 0 to 0`) do not appear explicitly in the following views but are left implicit. These views are named by appending "0" to the name of the selected sort; for example, the standard view from `DEFAULT` into `RAT` selecting the sort `Rat` and `0` as the default element is named `Rat0`.

```
view Nat0 from DEFAULT to NAT is
  sort Elt to Nat .
endv

view Int0 from DEFAULT to INT is
  sort Elt to Int .
endv

view Rat0 from DEFAULT to RAT is
  sort Elt to Rat .
endv

view Float0 from DEFAULT to FLOAT is
  sort Elt to Float .
  op 0 to term 0.0 .
endv

view String0 from DEFAULT to STRING is
  sort Elt to String .
  op 0 to term "" .
endv

view Qid0 from DEFAULT to QID is
  sort Elt to Qid .
  op 0 to term ' .
endv
```

7.12.3 STRICT-WEAK-ORDER and STRICT-TOTAL-ORDER

Although in Section 6.3.6 we have defined the notion of sorted list as based on a totally ordered set of elements, we will see in Section 7.13.6 how to relax this requirement in two different ways. The first possibility is to consider a *partially* strictly ordered set where the incomparability relation is transitive, that is, if a is not comparable with b and b is not comparable with c with respect to the given order, then a and c are not comparable either. The predefined `STRICT-WEAK-ORDER` theory below specifies a strict partial order with this additional

requirement, a concept known as *strict weak order*. The second possibility is to consider a *total preorder*, as specified in Section 7.12.4 below.

Given a strict partial order $<$, that is, an irreflexive and transitive binary relation, we define the *incomparability* relation by $x \sim y$ iff both $x \not< y$ and $y \not< x$. Incomparability is symmetric by definition, and its reflexivity follows from the irreflexivity of $<$. Therefore, when we impose the additional requirement of transitivity of incomparability, we get that the relation $\sim$ for a strict weak order is an equivalence relation.

Notice that `STRICT-WEAK-ORDER`, as presented below, imports the theory `TRIV` and also (in `protecting` mode) the module `BOOL`. The three equations express the required properties (antisymmetry is derivable from irreflexivity and transitivity) of the binary relation `_<_` on the sort `Elt`, as is made explicit in the corresponding labels.

```
fth STRICT-WEAK-ORDER is
  protecting BOOL .
  including TRIV .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
  eq X < X = false [nonexec label irreflexive] .
  ceq X < Y or Y < X or Y < Z or Z < Y = true if X < Z or Z < X
    [nonexec label incomparability-transitive] .
endfth
```

The following theory extends the previous one with a totality requirement, thus specifying a strict total order. Under these conditions, the incomparability relation reduces to the identity (because any pair of different elements is comparable) and the transitivity of incomparability holds trivially.

```
fth STRICT-TOTAL-ORDER is
  including STRICT-WEAK-ORDER .
  vars X Y : Elt .
  ceq X = Y if X < Y = false /\ Y < X = false [nonexec label total] .
endfth
```

The theory `STRICT-TOTAL-ORDER` is a different presentation of the equivalent theory `STOSET` for strict total orders introduced in Section 6.3.1.

There is a view from `TRIV` to `STRICT-WEAK-ORDER` that forgets the order and its properties. The name of this view coincides with the name of the target theory.

```
view STRICT-WEAK-ORDER from TRIV to STRICT-WEAK-ORDER is
endv
```

The inclusion from the theory `STRICT-WEAK-ORDER` into `STRICT-TOTAL-ORDER` gives rise to another view, which is also called as the target theory.

```
view STRICT-TOTAL-ORDER from STRICT-WEAK-ORDER
  to STRICT-TOTAL-ORDER is
endv
```

The Maude library includes views that map from `STRICT-TOTAL-ORDER` to built-in data type modules by selecting the main sort and the standard strict total order between the corresponding elements, namely, the "less than" comparison between numbers and the lexicographic ordering between strings, as described in previous sections. Again, operator mappings that are the identity (in this case of the form `op _<_ to _<_`) do not appear explicitly in the following views, but are left implicit. These views are named by appending "`<`" to the name of the

selected sort; for example, the standard view from STRICT-TOTAL-ORDER into RAT is named Rat<.

```
view Nat< from STRICT-TOTAL-ORDER to NAT is
  sort Elt to Nat .
endv

view Int< from STRICT-TOTAL-ORDER to INT is
  sort Elt to Int .
endv

view Rat< from STRICT-TOTAL-ORDER to RAT is
  sort Elt to Rat .
endv

view Float< from STRICT-TOTAL-ORDER to FLOAT is
  sort Elt to Float .
endv

view String< from STRICT-TOTAL-ORDER to STRING is
  sort Elt to String .
endv
```

As explained in Section 6.3.2, these views impose some proof obligations corresponding in this case to the properties that are stated about the binary relation selected in the target module; recall that such proof obligations are not discharged or checked by the system.

7.12.4 TOTAL-PREORDER and TOTAL-ORDER

The predefined TOTAL-PREORDER theory specifies, as its name clearly suggests, a *total preorder*, that is, a total binary relation which is reflexive and transitive. This theory will also be used as requirement for sorting lists in Section 7.13.6.

The notions of strict weak order (see Section 7.12.3) and of total preorder are complementary: the set-theoretic complement of a strict weak order is a total preorder and vice versa. They can also be related in a way that preserves the direction of the order. Given a strict weak order $<$, a total preorder $\leq$ is obtained by defining $x \leq y$ whenever $y \not< x$. In the other direction, a strict weak order $<$ is obtained from a total preorder $\leq$ by defining $x < y$ whenever $y \not\leq x$.

Given a total preorder $\leq$, we say that two elements x and y are *equivalent* iff both $x \leq y$ and $y \leq x$. Then, it follows from the properties of a total preorder that this is an equivalence relation and, furthermore, two elements are equivalent in a total preorder if and only if they are incomparable in the associated strict weak order (we have seen in Section 7.12.3 that the incomparability relation $\sim$ associated to a strict weak order is an equivalence relation).

Both kinds of relations capture the notion that the set of elements is split into partitions which are linearly ordered. This situation naturally arises when records are compared on a given field.

The theory TOTAL-PREORDER, as presented below, imports the theory TRIV and the module BOOL. The three equations express the required properties of the binary relation _<=_ on the sort Elt.

```
fth TOTAL-PREORDER is
  protecting BOOL .
  including TRIV .
```

```
  op _<=_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X <= X = true [nonexec label reflexive] .
  ceq X <= Z = true if X <= Y /\ Y <= Z [nonexec label transitive] .
  eq X <= Y or Y <= X = true [nonexec label total] .
endfth
```

A total *order* is a total preorder that, in addition, is antisymmetric.

```
fth TOTAL-ORDER is
  inc TOTAL-PREORDER .
  vars X Y : Elt .
  ceq X = Y if X <= Y /\ Y <= X [nonexec label antisymmetric] .
endfth
```

The theory TOTAL-ORDER is a different presentation of the equivalent theory NSTOSET for *non-strict* total orders introduced in Section 6.3.1. Its name follows the usual convention according to which, when nothing is said, a total order is assumed to be reflexive, that is, non-strict.

There is a view from TRIV to TOTAL-PREORDER, named like the target theory, that forgets the binary relation and its preorder properties.

```
view TOTAL-PREORDER from TRIV to TOTAL-PREORDER is
endv
```

The following view represents the inclusion from the TOTAL-PREORDER theory into TOTAL-ORDER.

```
view TOTAL-ORDER from TOTAL-PREORDER to TOTAL-ORDER is
endv
```

In the Maude prelude we can also find views that map from TOTAL-ORDER to several built-in data type modules by selecting the main sort and the standard non-strict total order between the corresponding elements, namely, the "less than or equal to" comparison between numbers and the lexicographic ordering between strings. These views are named by appending "<=" to the name of the selected sort; for example, the standard view from TOTAL-ORDER into FLOAT is named Float<.

```
view Nat<= from TOTAL-ORDER to NAT is
  sort Elt to Nat .
endv

view Int<= from TOTAL-ORDER to INT is
  sort Elt to Int .
endv

view Rat<= from TOTAL-ORDER to RAT is
  sort Elt to Rat .
endv

view Float<= from TOTAL-ORDER to FLOAT is
  sort Elt to Float .
endv

view String<= from TOTAL-ORDER to STRING is
  sort Elt to String .
endv
```

Again, these views impose some proof obligations that are not discharged or checked by the system.

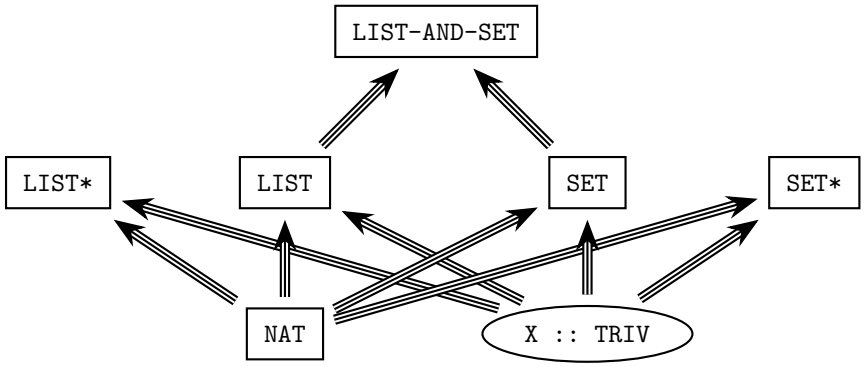


Figure 7.2: Importation graph of parameterized list and set modules

7.13 Containers: lists and sets

The current Maude prelude includes two parameterized containers: *lists* and *sets*.

Figure 7.2 shows the relationships between the modules described in this section specifying parameterized lists and sets, including the theory `TRIV`. The module specifying sortable lists is not included in this figure, because its relationship is more complex than `protecting` importations (see later Figure 7.4).

Other container data types may be added to the Maude prelude in the future.

7.13.1 Lists

Lists over a given sort of elements (provided by the theory `TRIV`) are constructed from the constant `nil` (representing the empty list) and singleton lists (identified with the corresponding elements by means of a subsort declaration) by means of an *associative* concatenation operator written as juxtaposition with empty syntax `__`.

Since there are several operations that are not well defined over the empty list, it is most useful to define the subsort of non-empty lists.

```
fmod LIST{X :: TRIV} is
  protecting NAT .
  sorts NeList{X} List{X} .
  subsort X$Elt < NeList{X} < List{X} .

  op nil : -> List{X} [ctor] .
  op __ : List{X} List{X} -> List{X} [ctor assoc id: nil prec 25] .
  op __ : NeList{X} List{X} -> NeList{X} [ctor ditto] .
  op __ : List{X} NeList{X} -> NeList{X} [ctor ditto] .

  vars E E' : X$Elt .
  vars A L : List{X} .
  var  C : Nat .
```

The operator `append` is just another name for concatenation.

```
  op append : List{X} List{X} -> List{X} .
  op append : NeList{X} List{X} -> NeList{X} .
  op append : List{X} NeList{X} -> NeList{X} .
```

```
eq append(A, L) = A L .
```

The operations `head` and `tail` take and discard, respectively, the first (leftmost) element in a list. Analogously, the operations `last` and `front` take and discard, respectively, the last (rightmost) element in a list. It is enough to have one equation for each operation, because the case of a singleton list is obtained by matching modulo identity with `L = nil`.

```
op head : NeList{X} -> X$Elt .
eq head(E L) = E .

op tail : NeList{X} -> List{X} .
eq tail(E L) = L .

op last : NeList{X} -> X$Elt .
eq last(L E) = E .

op front : NeList{X} -> List{X} .
eq front(L E) = L .
```

The predicate `occurs` checks whether an element appears in any position in a list. The two equations in its specification correspond to the typical case analysis (or structural induction) over lists: either the list is empty or we consider the corresponding first element (in the latter case, again one equation is enough).

```
op occurs : X$Elt List{X} -> Bool .
eq occurs(E, nil) = false .
eq occurs(E, E' L) = if E == E' then true else occurs(E, L) fi .
```

Reversing a list is accomplished by means of the operator `reverse`, which is efficiently defined through an auxiliary operator `$reverse` that has an additional *accumulator* argument. With this argument, `$reverse` has a simple *tail-recursive* and thus efficient definition.

```
op reverse : List{X} -> List{X} .
op reverse : NeList{X} -> NeList{X} .
eq reverse(L) = $reverse(L, nil) .

op $reverse : List{X} List{X} -> List{X} .
eq $reverse(nil, A) = A .
eq $reverse(E L, A) = $reverse(L, E A).
```

The tail-recursive method of definition just described will be used in the specification of several other operators, including the `size` operator on lists, which computes the number of elements in a list.

```
op size : List{X} -> Nat .
op size : NeList{X} -> NzNat .
eq size(L) = $size(L, 0) .

op $size : List{X} Nat -> Nat .
eq $size(nil, C) = C .
eq $size(E L, C) = $size(L, C + 1) .
endfm
```

In the Maude prelude there are two list instantiations on built-in data types (natural numbers and quoted identifiers) that are needed by the metalevel (see Chapter 17).

```
fmod NAT-LIST is
```

```
  protecting LIST{Nat} * (sort NeList{Nat} to NeNatList,
                          sort List{Nat} to NatList) .
endfm

fmod QID-LIST is
  protecting LIST{Qid} * (sort NeList{Qid} to NeQidList,
                          sort List{Qid} to QidList) .
endfm
```

Other instantiations can be built as desired. For example, we can use the view `Int` from `TRIV` to `INT`, and then test some reductions, as follows.

```
fmod INT-LIST is
  pr LIST{Int} .
endfm

Maude> red in INT-LIST : reverse(0 -1 2 -3 4 -5 6) .
result NeList{Int}: 6 -5 4 -3 2 -1 0

Maude> red occurs(7, 0 -1 2 -3 4 -5 6) .
result Bool: false

Maude> red size(0 -1 2 -3 4 -5 6) .
result NzNat: 7
```

7.13.2 Sets

Sets over a given sort of elements (provided by the theory `TRIV`) are built from the constant `empty` and singleton sets (identified with the corresponding elements by means of a subsort declaration) with an *associative*, *commutative*, and *idempotent* union operator written `_,_`. The first two such properties are declared as attributes, while the third is written as an equation; remember that the attributes `idem` and `assoc` cannot be used together (see Section 4.4.1).

```
fmod SET{X :: TRIV} is
  protecting EXT-BOOL .
  protecting NAT .
  sorts NeSet{X} Set{X} .
  subsort X$Elt < NeSet{X} < Set{X} .

  op empty : -> Set{X} [ctor] .
  op _,_ : Set{X} Set{X} -> Set{X}
      [ctor assoc comm id: empty prec 121 format (d r os d)] .
  op _,_ : NeSet{X} Set{X} -> NeSet{X} [ctor ditto] .

  var E : X$Elt .
  var N : NeSet{X} .
  vars A S S' : Set{X} .
  var C : Nat .

  eq N, N = N .
```

The prefix operator `union` is just another name for the infix operator `_,_`. Moreover, given the identification between elements and singleton sets, inserting an element is a particular case of union.

```
op union : Set{X} Set{X} -> Set{X} .
op union : NeSet{X} Set{X} -> NeSet{X} .
op union : Set{X} NeSet{X} -> NeSet{X} .
eq union(S, S') = S, S' .

op insert : X$Elt Set{X} -> Set{X} .
eq insert(E, S) = E, S .
```

The definitions of the operators `delete`, that deletes an element from a set, and `_in_`, that checks if an element belongs to a set, are based on the statement attribute `otherwise` (see Section 4.5.4):

1. When a given term representing a set matches the pattern `(E, S)` (modulo the equational attributes of the `_,_` operator), then we can delete the element `E` (and continue deleting, since there may be repetitions of such element in the given term), and state that indeed the element `E` belongs to the set.

2. *Otherwise*, the element `E` does not belong to the set and deleting such element does not change the set.

```
op delete : X$Elt Set{X} -> Set{X} .
eq delete(E, (E, S)) = delete(E, S) .
eq delete(E, S) = S [owise] .

op _in_ : X$Elt Set{X} -> Bool .
eq E in (E, S) = true .
eq E in S = false [owise] .
```

The operator `|_|` computes the cardinality of a set. Its definition goes through an auxiliary operator `$card` with an additional accumulator argument that allows a tail-recursive definition. In turn, the specification of `$card` is based on an equation that eliminates repetitions of elements in a term representing a set; when such equation can no longer be applied (hence the `owise` attribute in the last equation), the accumulator argument does its job by counting once each different element.

```
op |_| : Set{X} -> Nat .
op |_| : NeSet{X} -> NzNat .
eq | S | = $card(S, 0) .

op $card : Set{X} Nat -> Nat .
eq $card(empty, C) = C .
eq $card((N, N, S), C) = $card((N, S), C) .
eq $card((E, S), C) = $card(S, C + 1) [owise] .
```

Both the intersection and set difference operations also use an auxiliary operation with a tail-recursive efficient definition. The accumulator argument keeps the elements that belong to both sets (for intersection) or to the first but not to the second set (for difference).

```
op intersection : Set{X} Set{X} -> Set{X} .
eq intersection(S, empty) = empty .
eq intersection(S, N) = $intersect(S, N, empty) .

op $intersect : Set{X} Set{X} Set{X} -> Set{X} .
eq $intersect(empty, S', A) = A .
eq $intersect((E, S), S', A)
```

```
    = $intersect(S, S', if E in S' then E, A else A fi) .

  op _\_ : Set{X} Set{X} -> Set{X} [gather (E e)].
  eq S \ empty = S .
  eq S \ N = $diff(S, N, empty) .

  op $diff : Set{X} Set{X} Set{X} -> Set{X} .
  eq $diff(empty, S', A) = A .
  eq $diff((E, S), S', A)
    = $diff(S, S', if E in S' then A else E, A fi) .
```

The following two predicates check whether their first argument is a (proper) subset of the second argument. The second one is defined in terms of the first, and in both cases the corresponding equations use the short-circuit version `_and-then_` of conjunction imported from the `EXT-BOOL` module.

```
  op _subset_ : Set{X} Set{X} -> Bool .
  eq empty subset S' = true .
  eq (E, S) subset S' = E in S' and-then S subset S' .

  op _psubset_ : Set{X} Set{X} -> Bool .
  eq S psubset S' = S =/= S' and-then S subset S' .
endfm
```

The Maude metalevel (see Chapter 17) imports a set instantiation on the built-in data type of quoted identifiers.

```
fmod QID-SET is
  protecting SET{Qid} * (sort NeSet{Qid} to NeQidSet,
                         sort Set{Qid} to QidSet) .
endfm
```

Another example of instantiation with some reductions is the following:

```
fmod INT-SET is
  pr SET{Int} .
endfm

Maude> red in INT-SET : | -1, 2, -3, 3, 2, -1 | .
result NzNat: 4

Maude> red 4 in (-1, 2, -3, 3, 2, -1) .
result Bool: false

Maude> red insert(4, (-1, 2, -3, 3, 2, -1)) .
result NeSet{Int}: 2, 3, 4, -1, -3

Maude> red union((2, 3, 4, -1, -3, 0), (-1, 2, -3, 3, 2, -1)) .
result NeSet{Int}: 0, 2, 3, 4, -1, -3

Maude> red intersection((2, 3, 4, -1, -3, 0),
                        (-1, 2, -3, 3, 2, -1)) .
result NeSet{Int}: 2, 3, -1, -3

Maude> red (2, 3, 4, -1, -3, 0) \ (-1, 2, -3, 3, 2, -1) .
result NeSet{Int}: 0, 4
```

7.13.3 Relating lists and sets

The following module provides some operations that involve both lists and sets; since these data types are not affected by the new operations, both of them are imported in `protecting` mode.

```
fmod LIST-AND-SET{X :: TRIV} is
  protecting LIST{X} .
  protecting SET{X} .

  var E : X$Elt .
  vars A L : List{X} .
  var S : Set{X} .
```

The operation `makeSet` transforms a list into a set, that is, it forgets the order between the elements and its repetitions; operationally, it simply transforms the constructors `nil` and `__` for lists into the constructors `empty` and `_,_` for sets, but this is done in an efficient way by using an auxiliary operator `$makeSet` with an accumulator argument that allows a tail-recursive definition by structural induction on the list given as first argument. Notice that both operators are overloaded to take into account in their declarations whether their arguments are empty or not.

```
  op makeSet : List{X} -> Set{X} .
  op makeSet : NeList{X} -> NeSet{X} .
  eq makeSet(L) = $makeSet(L, empty) .

  op $makeSet : List{X} Set{X} -> Set{X} .
  op $makeSet : NeList{X} Set{X} -> NeSet{X} .
  op $makeSet : List{X} NeSet{X} -> NeSet{X} .
  eq $makeSet(nil, S) = S .
  eq $makeSet(E L, S) = $makeSet(L, (E, S)) .
```

An inverse operation `makeList` that transforms a set into a list will be seen in Section 7.13.7, because it only makes sense when we have additional information to put the elements of the set in a sequence in a univocally defined way.

The operations `filter` and `filterOut` take a list and a set as arguments, and return the list formed by those elements of the given list that belong and that do not belong, respectively, to the given set, in their original order. Again, both are defined by means of auxiliary operations with accumulator arguments allowing efficient tail-recursive definitions.

```
  op filter : List{X} Set{X} -> List{X} .
  eq filter(L, S) = $filter(L, S, nil) .

  op $filter : List{X} Set{X} List{X} -> List{X} .
  eq $filter(nil, S, A) = A .
  eq $filter(E L, S, A)
    = $filter(L, S, if E in S then A E else A fi) .

  op filterOut : List{X} Set{X} -> List{X} .
  eq filterOut(L, S) = $filterOut(L, S, nil) .

  op $filterOut : List{X} Set{X} List{X} -> List{X} .
  eq $filterOut(nil, S, A) = A .
  eq $filterOut(E L, S, A)
    = $filterOut(L, S, if E in S then A else A E fi) .
endfm
```

For illustration, we consider the following instantiation and some reductions.

```
fmod INT-LIST-AND-SET is
  pr LIST-AND-SET{Int} .
endfm

Maude> red in INT-LIST-AND-SET : filter((1 -1 1 -2 1), (1, 2)) .
result NeList{Int}: 1 1 1

Maude> red filterOut((1 -1 1 -2 1), (1, 2)) .
result NeList{Int}: -1 -2

Maude> red makeSet(1 -1 1 -2 1) .
result NeSet{Int}: 1, -1, -2
```

7.13.4 Generalized lists

With the construction of parameterized lists described in Section 7.13.1, we can build, for example, lists of integers, or lists of lists of integers, but we cannot build lists in which we have as elements both integers and lists of integers; for this, we specify in this section the container of *generalized* or *nestable lists*.

In this specification we cannot use empty syntax in the same way as in Section 7.13.1, because we need something to distinguish the different levels of nesting of lists inside lists. We use an auxiliary sort `Item`, whose data are both elements and generalized lists (see the subsort declarations below); then we put such items next to each other by juxtaposition, getting in this way data of another auxiliary sort `PreList`, and finally we put square brackets around a "prelist" in order to get a generalized list. Notice that there is no empty "prelist" and that the empty generalized list `[]` is declared separately.

```
fmod LIST*{X :: TRIV} is
  protecting NAT .
  sorts Item{X} PreList{X} NeList{X} List{X} .
  subsort X$Elt List{X} < Item{X} < PreList{X} .
  subsort NeList{X} < List{X} .

  op __ : PreList{X} PreList{X} -> PreList{X} [ctor assoc prec 25] .
  op [_] : PreList{X} -> NeList{X} [ctor] .
  op [] : -> List{X} [ctor] .

  vars A P : PreList{X} .
  var  L : List{X} .
  vars E E' : Item{X} .
  var  C : Nat .
```

The operator `append` now corresponds to concatenation of generalized lists and its definition is based on the juxtaposition of the "prelists" inside the generalized lists.

```
  op append : List{X} List{X} -> List{X} .
  op append : NeList{X} List{X} -> NeList{X} .
  op append : List{X} NeList{X} -> NeList{X} .
  eq append([], L) = L .
  eq append(L, []) = L .
  eq append([P], [A]) = [P A] .
```

The operations `head`, `tail`, `last`, and `front` work as for "standard" lists, but now they refer to the first or last *item* in the list, which can be either an element or a nested list. Now we need two equations for each operation, because the singleton case needs to be treated separately (recall that there is no empty "prelist").

```
op head : NeList{X} -> Item{X} .
eq head([E]) = E .
eq head([E P]) = E .

op tail : NeList{X} -> List{X} .
eq tail([E]) = [] .
eq tail([E P]) = [P] .

op last : NeList{X} -> Item{X} .
eq last([E]) = E .
eq last([P E]) = E .

op front : NeList{X} -> List{X} .
eq front([E]) = [] .
eq front([P E]) = [P] .
```

The predicate `occurs` checks whether an item (either an element or a list) appears in any position of the first level of a generalized list (but it does not go into deeper levels, that is, into nested lists). The three equations in its specification correspond to the typical case analysis (or structural induction) over these lists: either the list is empty, or it is a list with a single item, or it is a list with two or more items.

```
op occurs : Item{X} List{X} -> Bool .
eq occurs(E, []) = false .
eq occurs(E, [E']) = (E == E') .
eq occurs(E, [E' P])
  = if E == E' then true else occurs(E, [P]) fi .
```

The operators `reverse` and `size` for generalized lists work in a similar way to the operators with the same names in Section 7.13.1, and they are also defined by means of auxiliary operators `$reverse` and `$size`, respectively, with a tail-recursive definition. Notice, however, that these auxiliary operators work on "prelists" instead of lists. Moreover, `size` counts the number of items in the first level of a generalized list, but it does not count the items inside nested lists at deeper levels.

```
op reverse : List{X} -> List{X} .
op reverse : NeList{X} -> NeList{X} .
eq reverse([]) = [] .
eq reverse([E]) = [E] .
eq reverse([E P]) = [$reverse(P, E)] .

op $reverse : PreList{X} PreList{X} -> PreList{X} .
eq $reverse(E, A) = E A .
eq $reverse(E P, A) = $reverse(P, E A).

op size : List{X} -> Nat .
op size : NeList{X} -> NzNat .
eq size([]) = 0 .
eq size([P]) = $size(P, 0) .
```

```
  op $size : PreList{X} Nat -> NzNat .
  eq $size(E, C) = C + 1 .
  eq $size(E P, C) = $size(P, C + 1) .
endfm
```

We consider the following instantiation and sample reductions:

```
fmod INT-LIST* is
  pr LIST*{Int} .
endfm

Maude> red in INT-LIST* : append([1 []], [[] 2]) .
result NeList{Int}: [1 [] [] 2]

Maude> red reverse([[1 []] [[] 2]]) .
result NeList{Int}: [[[] 2] [1 []]]

Maude> red occurs(1, [[[] 2] [1 []]]) .
result Bool: false

Maude> red size([[[] 2] [1 []]]) .
result NzNat: 2
```

7.13.5 Generalized sets

The construction of generalized or nestable sets follows exactly the same pattern as the one we have seen for generalized lists in the previous section, but now we use braces instead of square brackets to make explicit the level of nesting. In particular, there is no empty "preset." Note that braces `{_}` and comma `_,_` exactly reflect standard set theory notation.

Notice that the sort named `Element` plays here the same role as `Item` played for nestable lists; do not confuse this sort with the sort `Elt` coming from the parameter theory `TRIV` in the form `X$Elt`.

The module `SET*` provides for generalized sets the same operations we have seen in Section 7.13.2 for "standard" sets, and, in addition, it specifies a powerset operator that was not possible in the previous setting.

```
fmod SET*{X :: TRIV} is
  protecting EXT-BOOL .
  protecting NAT .
  sorts Element{X} PreSet{X} NeSet{X} Set{X} .
  subsort X$Elt Set{X} < Element{X} < PreSet{X} .
  subsort NeSet{X} < Set{X} .

  op _,_ : PreSet{X} PreSet{X} -> PreSet{X}
       [ctor assoc comm prec 121 format (d r os d)] .
  op {_} : PreSet{X} -> NeSet{X} [ctor] .
  op {} : -> Set{X} [ctor] .

  vars P Q : PreSet{X} .
  vars A S : Set{X} .
  var  E : Element{X} .
  var  N : NeSet{X} .
  var  C : Nat .
```

```
eq {P, P} = {P} .
eq {P, P, Q} = {P, Q} .
```

The operations for insertion, deletion, and membership testing now work for items that can be either basic elements or nested sets, but always at the first level of nesting. For example, the membership predicate `_in_` cannot be used to test if a basic element belongs to a set inside another set, but on the other hand can check if a set is a member of another set. In other words, the operation `_in_` exactly corresponds to the set theory membership predicate $\in$. As in Section 7.13.2, the operators `delete` and `_in_` are defined by means of the `otherwise` attribute. Moreover, each one has an additional equation for the singleton case, which is treated separately because there is no empty "preset."

```
op insert : Element{X} Set{X} -> Set{X} .
eq insert(E, {}) = {E} .
eq insert(E, {P}) = {E, P} .

op delete : Element{X} Set{X} -> Set{X} .
eq delete(E, {E}) = {} .
eq delete(E, {E, P}) = delete(E, {P}) .
eq delete(E, S) = S [owise] .

op _in_ : Element{X} Set{X} -> Bool .
eq E in {E} = true .
eq E in {E, P} = true .
eq E in S = false [owise] .
```

The cardinality operator `|_|` computes the number of items (either basic elements or other sets, at the first level of nesting) in a given set. It is defined with the help of an auxiliary tail-recursive operator `$card` on "presets."

```
op |_| : Set{X} -> Nat .
op |_| : NeSet{X} -> NzNat .
eq | {} | = 0 .
eq | {P} | = $card(P, 0) .

op $card : PreSet{X} Nat -> Nat .
eq $card(E, C) = C + 1 .
eq $card((N, N, P), C) = $card((N, P), C) .
eq $card((E, P), C) = $card(P, C + 1) [owise] .
```

The union operator `union` on generalized sets is based on the "union" operator `_,_` on the "presets" inside the generalized sets.

```
op union : Set{X} Set{X} -> Set{X} .
op union : NeSet{X} Set{X} -> NeSet{X} .
op union : Set{X} NeSet{X} -> NeSet{X} .
eq union({}, S) = S .
eq union(S, {}) = S .
eq union({P}, {Q}) = {P, Q} .
```

The intersection and set difference operations for generalized sets have a specification very similar to the one seen in Section 7.13.2, including the use of tail-recursive auxiliary operations on "presets".

```
op intersection : Set{X} Set{X} -> Set{X} .
eq intersection({}, S) = {} .
```

```
  eq intersection(S, {}) = {} .
  eq intersection({P}, N) = $intersect(P, N, {}) .

  op $intersect : PreSet{X} Set{X} Set{X} -> Set{X} .
  eq $intersect(E, S, A) = if E in S then insert(E, A) else A fi .
  eq $intersect((E, P), S, A)
    = $intersect(P, S, $intersect(E, S, A)) .

  op _\_ : Set{X} Set{X} -> Set{X} [gather (E e)] .
  eq {} \ S = {} .
  eq S \ {} = S .
  eq {P} \ N = $diff(P, N, {}) .

  op $diff : PreSet{X} Set{X} Set{X} -> Set{X} .
  eq $diff(E, S, A) = if E in S then A else insert(E, A) fi .
  eq $diff((E, P), S, A) = $diff(P, S, $diff(E, S, A)) .
```

The powerset `2^ X` of a set `X` is computed by case analysis on the set `X`: it is either the empty set `{}` or a singleton set `{E}`, or it has two or more items `{E, P}`. In the last case we compute the total powerset `2^ X` by computing first the powerset `2^{P}` of the set without item `E` and then the union of this powerset `2^{P}` with the result of inserting the distinguished item `E` into all the items in the same powerset `2^{P}`. The last process is done by means of an auxiliary operation `$augment`.

```
  op 2^_ : Set{X} -> Set{X} .
  eq 2^{} = {{}} .
  eq 2^{E} = {{}, {E}} .
  eq 2^{E, P} = union(2^{P}, $augment(2^{P}, E, {})) .

  op $augment : NeSet{X} Element{X} Set{X} -> Set{X} .
  eq $augment({S}, E, A) = insert(insert(E, S), A) .
  eq $augment({S, P}, E, A)
    = $augment({P}, E, $augment({S}, E, A)) .
```

The specification of the subset predicates that check whether a set is included in another is completely analogous to the specification of the corresponding operations in Section 7.13.2.

```
  op _subset_ : Set{X} Set{X} -> Bool .
  eq {} subset S = true .
  eq {E} subset S = E in S .
  eq {E, P} subset S = E in S and-then {P} subset S .

  op _psubset_ : Set{X} Set{X} -> Bool .
  eq A psubset S = A =/= S and-then A subset S .
dfm
```

We consider the following instantiation and sample reductions:

```
fmod QID-SET* is
  pr SET*{Qid} .
endfm

Maude> red in QID-SET* : {'a} in {{'a}, {'b}, {'a, 'b}} .
result Bool: true

Maude> red | {{'a}, {'b}, {'a, 'b}} | .
result NzNat: 3
```

```
Maude> red union({{'a}, {'b}}, {{'a, 'b}}) .
result NeSet{Qid}: {{'a}, {'b}, {'a, 'b}}

Maude> red intersection({{'a}, {'b}}, {{'a, 'b}}) .
result Set{Qid}: {}

Maude> red 2^ {'a, 'b, 'c, 'd} .
result NeSet{Qid}:
  {{}, {'a}, {'b}, {'c}, {'d}, {'a, 'b}, {'a, 'c}, {'a, 'd},
   {'b, 'c}, {'b, 'd}, {'c, 'd}, {'a, 'b, 'c}, {'a, 'b, 'd},
   {'a, 'c, 'd}, {'b, 'c, 'd}, {'a, 'b, 'c, 'd}}
```

7.13.6 Sortable lists

In Section 6.3.6 we defined the notion of sorted list requiring a totally ordered set of elements, but this requirement can be relaxed. In principle, it is enough to have a transitive and antisymmetric order $<$ on a set E of elements (which are the requirements in the theory `TAOSET` from Section 6.3.1) to be able to define a *sorted list* L over E as a list such that for every pair (u, v) of members in L with u occurring before v and with $u \neq v$, it is the case that $v < u$ is false. However, in what follows we are not interested in defining sorted lists, but in specifying a sorting algorithm (more specifically, the *mergesort* algorithm) in a deterministic way. We require the sorting algorithm to be *stable*, so that incomparable elements remain in the same relative order as in the list provided as argument. For this notion to be well defined, we need to require either a strict weak order or a total preorder.

Sorting lists with respect to a strict weak order

Assume first that $<$ is a *strict weak order* over a set E, that is, a strict partial order with a transitive incomparability relation, which are precisely the requirements in the predefined theory `STRICT-WEAK-ORDER` of Section 7.12.3.

In order to define a *stable sorting* of a list L of elements over E, we consider each element of the list L as a pair (x, i), where x is the value of the element in E and i is the number indicating the position of x in L. We define an ordering $\ll$ on such pairs as follows: $(x, i) \ll (y, j)$ iff either $x < y$ or ($x \sim y$ and $i < j$). Then, it follows from the properties of $<$ and $\sim$ that $\ll$ is a strict total order, i.e., it is irreflexive, transitive, and total.

We can now define the stable sorting under $<$ of a list $e_1, e_2, \ldots, e_n$ of elements from E as follows: Take the list $(e_1, 1), (e_2, 2), \ldots, (e_n, n)$, find its unique ordering $(e_{s_1}, s_1), (e_{s_2}, s_2), \ldots, (e_{s_n}, s_n)$ under $\ll$, and output $e_{s_1}, e_{s_2}, \ldots, e_{s_n}$.

The parameterized module `WEAKLY-SORTABLE-LIST`, that specifies a stable version of mergesort on lists, imports "standard" lists (from Section 7.13.1), but first it is necessary to match the parameter theory `TRIV` of lists with the parameter theory `STRICT-WEAK-ORDER`. This is accomplished by means of the predefined view `STRICT-WEAK-ORDER` from `TRIV` to `STRICT-WEAK-ORDER` that forgets the order and its properties (see Section 7.12.3). A renaming is also applied to this instantiation in order to have more convenient sort names. This process is illustrated in the diagram of Figure 7.3, where `STRICT-WEAK-ORDER` has been abbreviated to `S-W-O`, the sort renaming has been abbreviated to α, and where the different types of arrows represent the different relationships between modules: importation (triple arrow), views between theories (dashed arrow named `S-W-O`), instantiation (dashed arrow), and renaming (dotted arrow named `_*(`α`)`, meaning the renaming whose second argument is α and whose first argument is still unknown).

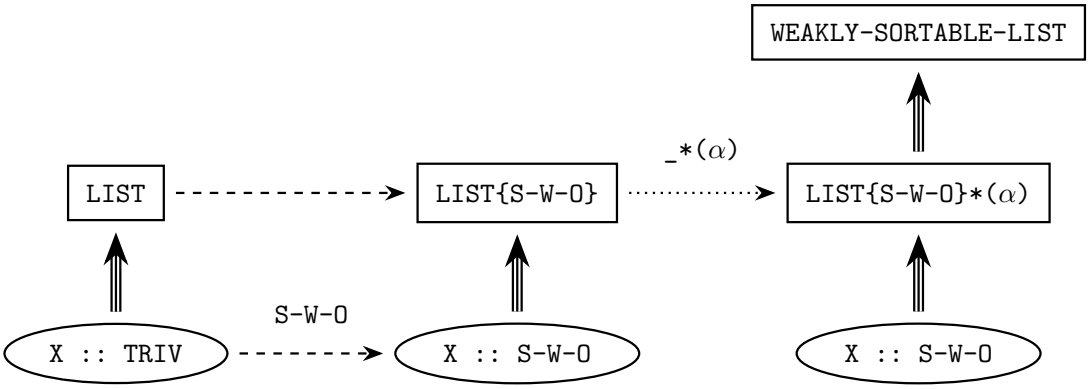


Figure 7.3: From lists to weakly sortable lists

```
fmod WEAKLY-SORTABLE-LIST{X :: STRICT-WEAK-ORDER} is
  pr LIST{STRICT-WEAK-ORDER}{X}
       * (sort NeList{STRICT-WEAK-ORDER}{X} to NeList{X},
          sort List{STRICT-WEAK-ORDER}{X} to List{X}) .
  sort $Split{X} .

  vars E E' : X$Elt .
  vars A A' L L' : List{X} .
  var  N : NeList{X} .
```

The main operation in this module is `sort`, that sorts a given list.[5] It is defined by case analysis on the list: if it is either the empty list or a singleton list, then it is already sorted; otherwise, we split the given list into two sublists, recursively sort both of them, and then merge the sorted results in order to obtain the final sorted list. This process is accomplished by means of three auxiliary operations, whose names are self-explanatory: `$split` (for the splitting, with an auxiliary result sort `$Split`), `$sort` (for the recursive sorting calls), and `$merge` (for the final merging).

```
  op sort : List{X} -> List{X} .
  op sort : NeList{X} -> NeList{X} .
  eq sort(nil) = nil .
  eq sort(E) = E .
  eq sort(E N) = $sort($split(E N, nil, nil)) .

  op $sort : $Split{X} -> List{X} .
  eq $sort($split(nil, L, L')) = $merge(sort(L), sort(L'), nil) .
```

The auxiliary operation `$split` has three arguments: the first one is the list to be split and the other two are accumulators (initially both empty) that keep the elements as they are moved from the main list into the appropriate sublists. In this way, we have an efficient tail-recursive definition.

```
  op $split : List{X} List{X} List{X} -> $Split{X} [ctor] .
  eq $split(E, A, A') = $split(nil, A E, A') .
  eq $split(E L E', A, A') = $split(L, A E, E' A') .
```

[5] We realize that terminology here can be a bit confusing, because in Maude `sort` is also a keyword for types.

The auxiliary operation `$merge` also has three arguments, but now the first two are the lists to be merged and the third one is the accumulator where the result is incrementally computed by means of another efficient tail-recursive definition.

The module also provides an operation `merge` that simply calls the previous operation with the empty accumulator. Notice that if both lists are sorted then the result of calling `merge` on them is a sorted list, but in general `merge` is a total function that can be called on any two lists whatsoever.

```
  op merge : List{X} List{X} -> List{X} .
  op merge : NeList{X} List{X} -> NeList{X} .
  op merge : List{X} NeList{X} -> NeList{X} .
  eq merge(L, L') = $merge(L, L', nil) .

  op $merge : List{X} List{X} List{X} -> List{X} .
  eq $merge(L, nil, A) = A L .
  eq $merge(nil, L, A) = A L .
  eq $merge(E L, E' L', A)
    = if E' < E
      then $merge(E L, L', A E')
      else $merge(L, E' L', A E)
      fi .
endfm
```

The Maude prelude also provides another predefined module for sorting lists, namely, `SORTABLE-LIST`, where the required order is strict and total, as specified in the predefined theory `STRICT-TOTAL-ORDER` of Section 7.12.3. Since the theory `STRICT-TOTAL-ORDER` is a strengthening of `STRICT-WEAK-ORDER` with the additional requirement of totality, we can use it as a parameter theory to specialize our `WEAKLY-SORTABLE-LIST` module to strict total orders, thus getting the `SORTABLE-LIST` module. For this we need a view from the theory `STRICT-WEAK-ORDER` into the theory `STRICT-TOTAL-ORDER`, which is precisely the predefined inclusion view `STRICT-TOTAL-ORDER` in Section 7.12.3.

Moreover, since we also use another renaming to have more convenient sort names, the construction of the parameterized module `SORTABLE-LIST` on top of `WEAKLY-SORTABLE-LIST` mirrors the process of constructing `WEAKLY-SORTABLE-LIST` on top of `LIST`, as described in Figure 7.4, where the sort renaming has been abbreviated to α', `WEAKLY-SORTABLE-LIST` to `W-S-LIST`, `STRICT-WEAK-ORDER` to `S-W-O`, and `STRICT-TOTAL-ORDER` to `S-T-O`. The reader should compare this figure with Figure 7.3 to appreciate the similarity between both.

```
fmod SORTABLE-LIST{X :: STRICT-TOTAL-ORDER} is
  pr WEAKLY-SORTABLE-LIST{STRICT-TOTAL-ORDER}{X}
       * (sort NeList{STRICT-TOTAL-ORDER}{X} to NeList{X},
          sort List{STRICT-TOTAL-ORDER}{X} to List{X}) .
endfm
```

We can use the predefined view `String<` from `STRICT-TOTAL-ORDER` to `String` (where `<` is the lexicographic order on strings) to instantiate the previous module before doing some sample reductions.

```
fmod STRING-SORTABLE-LIST is
  pr SORTABLE-LIST{String<} .
endfm

Maude> red in STRING-SORTABLE-LIST :
        $split("a" "quick" "brown" "fox" "jumps"
```

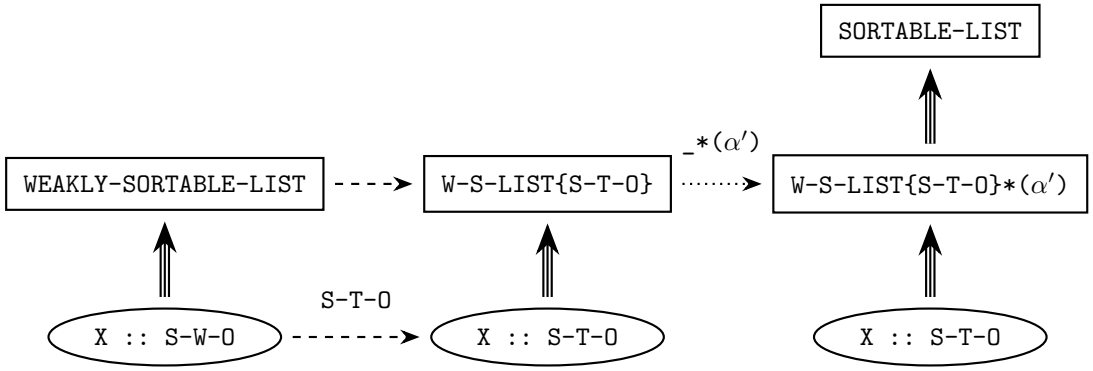


Figure 7.4: From weakly sortable lists to sortable lists

```
              "over" "the" "lazy" "dog", nil, nil) .
result $Split{STRICT-TOTAL-ORDER}{String<}:
  $split(nil,
         "a" "quick" "brown" "fox" "jumps",
         "over" "the" "lazy" "dog")

Maude> red merge("a" "quick" "brown" "fox" "jumps",
                 "over" "the" "lazy" "dog") .
result NeList{String<}:
  "a" "over" "quick" "brown" "fox" "jumps" "the" "lazy" "dog"

Maude> red sort("a" "quick" "brown" "fox" "jumps"
                "over" "the" "lazy" "dog") .
result NeList{String<}:
  "a" "brown" "dog" "fox" "jumps" "lazy" "over" "quick" "the"

Maude> red sort("a" "quick" "brown" "fox" "jumps" "over" "the"
                "lazy" "dog" "a" "quick" "brown" "fox" "jumps"
                "over" "the" "lazy" "dog") .
result NeList{String<}: "a" "a" "brown" "brown" "dog" "dog" "fox"
  "fox" "jumps" "jumps" "lazy" "lazy" "over" "over" "quick" "quick"
  "the" "the"
```

Sorting lists with respect to a total preorder

Assume now that $\leq$ is a *total preorder* over a set E, that is, a binary relation satisfying the requirements in the predefined theory `TOTAL-PREORDER` of Section 7.12.4.

To define a stable sorting of a list L of elements over E, we consider again each element of the list L as a pair (x, i), where x is the value of the element in E and i is the number indicating the position of x in L. We define an ordering $\ll$ on such pairs as follows, where now the definition of $\ll$ is slightly different given the non-strict nature of total preorders: $(x, i) \ll (y, j)$ iff either $y \not\leq x$ or ($x \leq y$ and $i \leq j$). Then, the properties of $\leq$ imply that $\ll$ is a (non-strict) total order, i.e., it is reflexive, antisymmetric, transitive, and total. From this, the definition of a stable sorting under $\leq$ of a list $e_1, e_2, \ldots, e_n$ of elements from E follows exactly the same steps as before: Take the list $(e_1, 1), (e_2, 2), \ldots, (e_n, n)$, find its unique ordering $(e_{s_1}, s_1), (e_{s_2}, s_2), \ldots, (e_{s_n}, s_n)$ under $\ll$, and output $e_{s_1}, e_{s_2}, \ldots, e_{s_n}$.

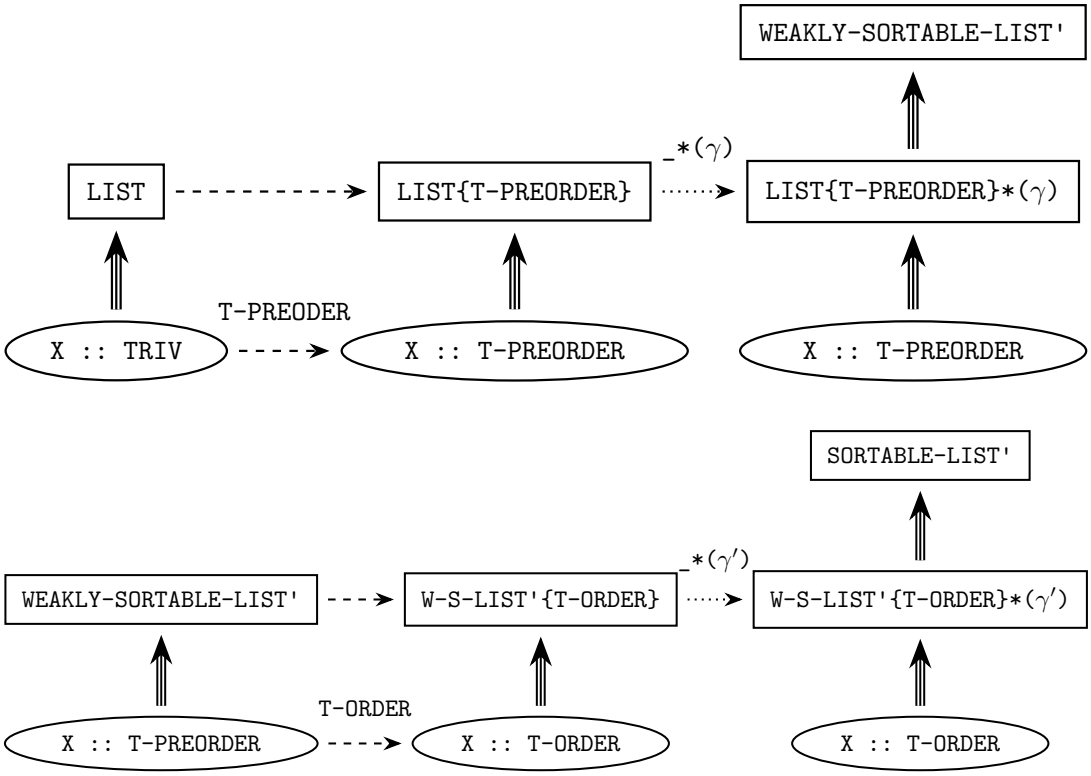


Figure 7.5: Another version of sortable lists

The following modules WEAKLY-SORTABLE-LIST' and SORTABLE-LIST' specify the merge-sort algorithm with respect to a total preorder and a (non-strict) total order, respectively. Their structure is completely analogous to the structure of WEAKLY-SORTABLE-LIST and SORTABLE-LIST already explained above. It is described in the two diagrams of Figure 7.5, where the sort renamings have been abbreviated to γ and γ', TOTAL-PREORDER to T-PREORDER, TOTAL-ORDER to T-ORDER, and WEAKLY-SORTABLE-LIST' W-S-LIST'.

```
fmod WEAKLY-SORTABLE-LIST'{X :: TOTAL-PREORDER} is
  pr LIST{TOTAL-PREORDER}{X}
       * (sort NeList{TOTAL-PREORDER}{X} to NeList{X},
          sort List{TOTAL-PREORDER}{X} to List{X}) .
  sort $Split{X} .

  vars E E' : X$Elt .
  vars A A' L L' : List{X} .
  var  N : NeList{X} .

  op sort : List{X} -> List{X} .
  op sort : NeList{X} -> NeList{X} .
  eq sort(nil) = nil .
  eq sort(E) = E .
  eq sort(E N) = $sort($split(E N, nil, nil)) .
```

```
  op $sort : $Split{X} -> List{X} .
  eq $sort($split(nil, L, L')) = $merge(sort(L), sort(L'), nil) .

  op $split : List{X} List{X} List{X} -> $Split{X} [ctor] .
  eq $split(E, A, A') = $split(nil, A E, A') .
  eq $split(E L E', A, A') = $split(L, A E, E' A') .

  op merge : List{X} List{X} -> List{X} .
  op merge : NeList{X} List{X} -> NeList{X} .
  op merge : List{X} NeList{X} -> NeList{X} .
  eq merge(L, L') = $merge(L, L', nil) .

  op $merge : List{X} List{X} List{X} -> List{X} .
  eq $merge(L, nil, A) = A L .
  eq $merge(nil, L, A) = A L .
  eq $merge(E L, E' L', A)
    = if E <= E'
      then $merge(L, E' L', A E)
      else $merge(E L, L', A E')
      fi .
endfm

fmod SORTABLE-LIST'{X :: TOTAL-ORDER} is
  pr WEAKLY-SORTABLE-LIST'{TOTAL-ORDER}{X}
       * (sort NeList{TOTAL-ORDER}{X} to NeList{X},
          sort List{TOTAL-ORDER}{X} to List{X}) .
endfm
```

Apart from the changes in the requirement theories and the module names, the main difference bewteen both approaches appears in the third `$merge` equation. In the `WEAKLY-SORTABLE-LIST` module we have

```
eq $merge(E L, E' L', A)
  = if E' < E
    then $merge(E L, L', A E')
    else $merge(L, E' L', A E)
    fi .
```

Here we are dealing with a strict weak order. We test `E' < E`. If it is true, then by irreflexivity we know that `E < E'` is false, and the element `E'` from the second list is appended to the merged list. Whereas if `E' < E` is false, we know that either `E < E'` holds or `E` and `E'` are incomparable. Either way, the element `E` from the first list is appended to the merged list, either because it is smaller or because it is incomparable and we are preserving the original relative positions in the list (stability).

On the other hand, in the `WEAKLY-SORTABLE-LIST'` module we have

```
eq $merge(E L, E' L', A)
  = if E <= E'
    then $merge(L, E' L', A E)
    else $merge(E L, L', A E')
    fi .
```

In this case we are dealing with a total preorder. We test `E <= E'`. If it is true, then either `E' <= E` is false or `E` and `E'` are equivalent. Either way, the element `E` from the first list is

appended to the merged list, either because it is smaller or because it is equivalent and we are preserving the original relative positions in the list (stability). If `E <= E'` is false, then `E' <= E` holds by totality and therefore `E'` is appended to the merged list.

We can redo with these modules the same instantiation we considered above, but using now the predefined view `String<=` from `TOTAL-ORDER` to `String`, where `<=` is the non-strict lexicographic order on strings.

```
fmod STRING-SORTABLE-LIST' is
  pr SORTABLE-LIST'{String<=} .
endfm

Maude> red in STRING-SORTABLE-LIST' :
        sort("a" "quick" "brown" "fox" "jumps"
             "over" "the" "lazy" "dog") .
result NeList{String<=}:
  "a" "brown" "dog" "fox" "jumps" "lazy" "over" "quick" "the"

Maude> red sort("a" "quick" "brown" "fox" "jumps" "over" "the"
                "lazy" "dog" "a" "quick" "brown" "fox" "jumps"
                "over" "the" "lazy" "dog") .
result NeList{String<=}: "a" "a" "brown" "brown" "dog" "dog" "fox"
  "fox" "jumps" "jumps" "lazy" "lazy" "over" "over" "quick" "quick"
  "the" "the"
```

7.13.7 Making lists out of sets

In Section 7.13.3 we have seen an operation `makeSet` that transforms a list into a set with the same elements. On the other hand, transforming a set into a list imposes some order on the given elements, which can be done in many different ways, and therefore only makes sense as a function when we have additional information over those elements that allows us to choose a unique sequence. The solution adopted here is to require either a strict or a non-strict total order on the elements, so that the resulting list is the corresponding sorted list. For this we use the `sort` operation defined either in the `SORTABLE-LIST` module or in the `SORTABLE-LIST'` module described in the previous section. In both versions the main operation `makeList` is defined in terms of an auxiliary operation `$makeList` with an accumulator in order to have a more efficient definition.

In both versions the `LIST-AND-SET` module is imported with a double renaming (different in each case), which is needed for correct sharing of a renamed copy of the `LIST` module, because Core Maude does not evaluate the composition of renamings but applies them sequentially. If we computed manually and used this simpler renaming, we would get a different renaming of `LIST` imported by each `protecting` declaration; then, while these renamings would have the same effect, we would import two renamed copies of `LIST` rather than a shared copy.

This is the first version, using a strict total order.

```
fmod SORTABLE-LIST-AND-SET{X :: STRICT-TOTAL-ORDER} is
  pr SORTABLE-LIST{X} .
  pr LIST-AND-SET{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
       * (sort NeList{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
            to NeList{STRICT-TOTAL-ORDER}{X},
          sort List{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
            to List{STRICT-TOTAL-ORDER}{X})
       * (sort NeList{STRICT-TOTAL-ORDER}{X} to NeList{X},
```

```
          sort List{STRICT-TOTAL-ORDER}{X} to List{X},
          sort NeSet{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
            to NeSet{X},
          sort Set{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
            to Set{X}) .

  var E : X$Elt .
  var L : List{X} .
  var S : Set{X} .

  op makeList : Set{X} -> List{X} .
  op makeList : NeSet{X} -> NeList{X} .
  eq makeList(S) = $makeList(S, nil) .

  op $makeList : Set{X} List{X} -> List{X} .
  op $makeList : NeSet{X} List{X} -> NeList{X} .
  op $makeList : Set{X} NeList{X} -> NeList{X} .
  eq $makeList((E, E, S), L) = $makeList((E, S), L) .
  eq $makeList((E, S), L) = $makeList(S, E L) [owise] .
endfm
```

Notice that `makeList` is only a partial inverse to `makeSet`, not only because of sorting the elements, but also because in a set repetitions do not matter. In general, for a set `S` and a list `L` we have `makeSet(makeList(S))` = `S`, but in general `makeList(makeSet(L))` ≠ `L`.

We consider an instantiation with the predefined view `Int<` and some sample reductions.

```
fmod INT-SORTABLE-LIST-AND-SET is
  pr SORTABLE-LIST-AND-SET{Int<} .
endfm
```

Notice that in the following first reduction we get a list different from the original one, while in the second reduction we get a different representation (where repetitions have been eliminated) of the same set. Those possible repetitions are already eliminated before producing the corresponding list, as shown in the third reduction.

```
Maude> red in INT-SORTABLE-LIST-AND-SET :
        makeList(makeSet(1 -1 1 -2 1 0)) .
result NeList{Int<}: -2 -1 0 1

Maude> red makeSet(makeList((5, 4, 3, 4, 5))) .
result NeSet{Int<}: 3, 4, 5

Maude> red makeList((5, 4, 3, 4, 5)) .
result NeList{Int<}: 3 4 5
```

This is the second version, using a non-strict total order.

```
fmod SORTABLE-LIST-AND-SET'{X :: TOTAL-ORDER} is
  pr SORTABLE-LIST'{X} .
  pr LIST-AND-SET{TOTAL-PREORDER}{TOTAL-ORDER}{X}
       * (sort NeList{TOTAL-PREORDER}{TOTAL-ORDER}{X}
            to NeList{TOTAL-ORDER}{X},
          sort List{TOTAL-PREORDER}{TOTAL-ORDER}{X}
            to List{TOTAL-ORDER}{X})
       * (sort NeList{TOTAL-ORDER}{X} to NeList{X},
          sort List{TOTAL-ORDER}{X} to List{X},
```

```
        sort NeSet{TOTAL-PREORDER}{TOTAL-ORDER}{X} to NeSet{X},
        sort Set{TOTAL-PREORDER}{TOTAL-ORDER}{X} to Set{X}) .

  var E : X$Elt .
  var L : List{X} .
  var S : Set{X} .

  op makeList : Set{X} -> List{X} .
  op makeList : NeSet{X} -> NeList{X} .
  eq makeList(S) = $makeList(S, nil) .

  op $makeList : Set{X} List{X} -> List{X} .
  op $makeList : NeSet{X} List{X} -> NeList{X} .
  op $makeList : Set{X} NeList{X} -> NeList{X} .
  eq $makeList(empty, L) = sort(L) .
  eq $makeList((E, E, S), L) = $makeList((E, S), L) .
  eq $makeList((E, S), L) = $makeList(S, E L) [owise] .
endfm
```

We redo the same instantiation, now with the non-strict total order on integers.

```
fmod INT-SORTABLE-LIST-AND-SET' is
  pr SORTABLE-LIST-AND-SET'{Int<=} .
endfm

Maude> red in INT-SORTABLE-LIST-AND-SET' :
        makeList(makeSet(1 -1 1 -2 1 0)) .
result NeList{Int<=}: -2 -1 0 1

Maude> red makeSet(makeList((5, 4, 3, 4, 5))) .
result NeSet{Int<=}: 3, 4, 5

Maude> red makeList((5, 4, 3, 4, 5)) .
result NeList{Int<=}: 3 4 5
```

7.14 Maps and arrays

Both *maps* and *arrays* represent a function f between two sets as a set of pairs of the form

$$\{(a_1, f(a_1)), (a_2, f(a_2)), \ldots, (a_n, f(a_n))\}$$

in the graph of the function; each pair $(a_i, f(a_i))$ is called an *entry* in both cases.

The difference between maps and arrays is that the former leave undefined the result of f over those values not present in the set above, while the latter assign a "default" value result in that case.

However, notice that the modules below *do not check*, for efficiency reasons, that all values a_i in the first components of a set of pairs like the previous one are *different* (although the operations for insertion and look up make sure that the corresponding result is well defined). The situation of having a set of entries with repeated first components never arises if such a map or array is initially the empty one and then it is only modified by means of the `insert` operation. See Section 6.3.7 for a more careful specification of (finite) partial functions checking these requirements.

7.14.1 Maps

As explained above, a map is defined as a "set" (built with the associative and commutative operator `_,_`) of entries. Notice that `Entry`, whose only constructor is the operator `_|->_`, is a subsort of `Map`.

The domain and codomain values of the map come from the parameters of the parameterized data type, both of them satisfying the theory `TRIV` and thus providing a set of elements.

The module `MAP` provides a constant `undefined` of the *kind* `[Y$Elt]` corresponding to the sort `Y$Elt` and representing the undefined result.

```
fmod MAP{X :: TRIV, Y :: TRIV} is
  protecting BOOL .
  sorts Entry{X,Y} Map{X,Y} .
  subsort Entry{X,Y} < Map{X,Y} .

  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
  op empty : -> Map{X,Y} [ctor] .
  op _,_ : Map{X,Y} Map{X,Y} -> Map{X,Y}
       [ctor assoc comm id: empty prec 121 format (d r os d)] .
  op undefined : -> [Y$Elt] [ctor] .

  var  D : X$Elt .
  vars R R' : Y$Elt .
  var  M : Map{X,Y} .
```

The operator `insert` adds a new entry to a map, but when the first argument already appears in the domain of definition of the map, the second argument is used to *update* the map. Notice the use of matching and of the `otherwise` attribute to distinguish these two cases in a simple way. Furthermore, in the first case, an auxiliary operation `$hasMapping` is used to make sure that in the resulting map only one entry is associated with the given value. The operation `$hasMapping` checks whether a domain value actually has an associated entry in a map.

```
  op insert : X$Elt Y$Elt Map{X,Y} -> Map{X,Y} .
  eq insert(D, R, (M, D |-> R'))
    = if $hasMapping(M, D)
      then insert(D, R, M)
      else (M, D |-> R)
      fi .
  eq insert(D, R, M) = (M, D |-> R) [owise] .

  op $hasMapping : Map{X,Y} X$Elt -> Bool .
  eq $hasMapping((M, D |-> R), D) = true .
  eq $hasMapping(M, D) = false [owise] .
```

The lookup operator is represented with the notation `_[_]`. Again, matching and `owise` are used to distinguish whether or not the second argument appears in the domain of definition of the map provided as first argument. When the answer is affirmative an the map contains exactly one entry associated with such argument (as checked with the auxiliary operation `$hasMapping`), the result is the value provided in that entry. When the answer is negative or the map is not well defined because there is more than one entry associated with the same argument, the result is the constant `undefined` in the kind, with the self-explanatory meaning that in those cases the map is undefined on the given argument.

```
  op _[_] : Map{X,Y} X$Elt -> [Y$Elt] [prec 23] .
```

```
  eq (M, D |-> R)[D]
    = if $hasMapping(M, D) then undefined else R fi .
  eq M[D] = undefined [owise] .
endfm
```

We use the predefined views `String` and `Nat` (see Section 7.12.1) to define maps from strings to natural numbers, and do some sample reductions.

```
fmod STRING-NAT-MAP is
  pr MAP{String, Nat} .
endfm

Maude> red in STRING-NAT-MAP :
        insert("one", 1,
          insert("two", 2, insert("three", 3, empty))) .
result Map{String,Nat}: "one" |-> 1, "three" |-> 3, "two" |-> 2

Maude> red insert("one", 1,
            insert("two", 2,
              insert("three", 3, empty)))["two"] .
result NzNat: 2

Maude> red insert("one", 1,
            insert("two", 2,
              insert("three", 3, empty)))["four"] .
result [FindResult]: undefined

Maude> red ("a" |-> 3, "a" |-> 2)["a"] .
result [FindResult]: undefined
```

The last reduction shows that the undesired repetition of a domain value in two entries of the same map also produces the `undefined` constant as result.

7.14.2 Arrays

As explained above, arrays work like maps, with the difference that an attempt to look up an unmapped value always returns the default value, i.e., arrays have a *sparse array* behavior (hence the name). In the same spirit, mappings to the default value are never inserted.

The main difference between maps and arrays is already made explicit in the parameters of the parameterized data type: while the first one satisfies the theory `TRIV`, the second one satisfies the theory `DEFAULT` that in addition to a set of data provides a default value `0` (see Section 7.12.2).

The constructor for entries is named `_|->_`, as for maps, while the set constructor is denoted here `_;_`.

```
fmod ARRAY{X :: TRIV, Y :: DEFAULT} is
  protecting BOOL .
  sorts Entry{X,Y} Array{X,Y} .
  subsort Entry{X,Y} < Array{X,Y} .

  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
  op empty : -> Array{X,Y} [ctor] .
  op _;_ : Array{X,Y} Array{X,Y} -> Array{X,Y}
       [ctor assoc comm id: empty prec 71 format (d r os d)] .
```

```
  var  D : X$Elt .
  vars R R' : Y$Elt .
  var  A : Array{X,Y} .
```

The definition of the operator `insert` for arrays adds a check to the definition of the same operator for maps so that, as mentioned above, entries whose second value is the default value `0` are never inserted. Note, however, that mappings to the default value `0` that are created with the constructors `_|->_` and `_;_`, rather than the `insert` operator, are not removed as doing this check each time a new array is formed would be excessively inefficient. Furthermore, as we have already seen for maps, in the first case, an auxiliary operation `$hasMapping` is used to make sure that in the resulting array only one entry is associated with the given value.

```
  op insert : X$Elt Y$Elt Array{X,Y} -> Array{X,Y} .
  eq insert(D, R, (A ; D |-> R'))
    = if $hasMapping(A, D)
      then insert(D, R, A)
      else if R == 0 then A else (A ; D |-> R) fi
      fi .
  eq insert(D, R, A)
    = if R == 0 then A else (A ; D |-> R) fi [owise] .

  op $hasMapping : Array{X,Y} X$Elt -> Bool .
  eq $hasMapping((A ; D |-> R), D) = true .
  eq $hasMapping(A, D) = false [owise] .
```

The definition of the lookup operator for arrays only differs from the one for maps in the occurrence of the default value `0` instead of the constant `undefined`. Now, if an argument has more than one associated entry (as checked with the auxiliar operation `$hasMapping`), it is considered to be "unmapped" and the result is also the default value.

```
  op _[_] : Array{X,Y} X$Elt -> Y$Elt [prec 23] .
  eq (A ; D |-> R)[D]
    = if $hasMapping(A, D) then 0 else R fi .
  eq A[D] = 0 [owise] .
endfm
```

We do the same instantiation for arrays as for maps, with the predefined views `String` from Section 7.12.1 and `Nat0` from Section 7.12.2).

```
fmod STRING-NAT-ARRAY is
  pr ARRAY{String, Nat0} .
endfm

Maude> red in STRING-NAT-ARRAY :
        insert("one", 1,
          insert("two", 2, insert("three", 3, empty))) .
result Array{String,Nat0}: "one" |-> 1 ; "three" |-> 3 ; "two" |-> 2

Maude> red insert("one", 0,
            insert("two", 2, insert("three", 3, empty))) .
result Array{String,Nat0}: "three" |-> 3 ; "two" |-> 2

Maude> red insert("one", 1,
            insert("two", 2,
              insert("three", 3, empty)))["two"] .
result NzNat: 2
```

```
Maude> red insert("one", 1,
           insert("two", 2,
             insert("three", 3, empty)))["four"] .
result Zero: 0
```

7.15 A linear Diophantine equation solver

The Maude system includes a built-in linear Diophantine equation solver. The interface to the solver is defined in the file `linear.maude` which contains the functional module `DIOPHANTINE`. The current solver finds non-negative solutions of a system S of n simultaneous linear equations in m variables having the form $Mv = c$, where M is an $n \times m$ integer coefficient matrix, v is a column vector of m variables and c is a column vector of n integer constants.

Both matrices and vectors are represented as sparse arrays with their dimensions implicit and their indices starting from 0. For this we make heavy use of the parameterized module `ARRAY`, described in Section 7.14.2.

First, a data type of pairs of natural numbers to be used as indices for matrices is created.

```
fmod INDEX-PAIR is
  pr NAT .
  sort IndexPair .
  op _,_ : Nat Nat -> IndexPair [ctor] .
endfm
```

Then, we instantiate (and rename as desired) the parameterized module `ARRAY` to obtain matrices of integers. Notice that `Int0` is the view from `DEFAULT` to `INT` given in Section 7.12.2

```
view IndexPair from TRIV to INDEX-PAIR is
  sort Elt to IndexPair .
endv

fmod MATRIX{X :: DEFAULT} is
  pr (ARRAY * (sort Entry{X,Y} to Entry{Y},
               sort Array{X,Y} to Matrix{Y}))
     {IndexPair, X} .
endfm

fmod INT-MATRIX is
  pr MATRIX{Int0} * (sort Entry{Int0} to IntMatrixEntry,
                     sort Matrix{Int0} to IntMatrix,
                     op empty to zeroMatrix) .
endfm
```

For example, the matrices

$$\begin{pmatrix} 1 & 2 \\ 0 & -1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 2 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

are both represented by the same term

```
(0,0) |-> 1 ; (0,1) |-> 2 ; (1,1) |-> -1
```

Vectors are represented in a similar way as sparse arrays with natural numbers as indices. We use here the view `Int0` already mentioned above and also the view `Nat` from `TRIV` to `NAT` given in Section 7.12.1. The view `IntVector` defined below will be used to construct sets of vectors later on.

```
fmod VECTOR{X :: DEFAULT} is
  pr (ARRAY * (sort Entry{X,Y} to Entry{Y},
               sort Array{X,Y} to Vector{Y}))
     {Nat, X} .
endfm

fmod INT-VECTOR is
  pr VECTOR{Int0} * (sort Entry{Int0} to IntVectorEntry,
                     sort Vector{Int0} to IntVector,
                     op empty to zeroVector) .
endfm

view IntVector from TRIV to INT-VECTOR is
  sort Elt to IntVector .
endv
```

No distinction is made between row and column vectors, so, for example, both the row vector $\begin{pmatrix} -2 & 0 & 0 & 3 \end{pmatrix}$ and its transpose $\begin{pmatrix} -2 & 0 & 0 & 3 \end{pmatrix}^t$ are represented by the same term

```
0 |-> -2 ; 3 |-> 3
```

The constants `zeroMatrix` and `zeroVector` denote the all zero matrix and vector, respectively.

The main module `DIOPHANTINE` begins defining pairs of sets of integer vectors, as follows:

```
fmod DIOPHANTINE is
  pr STRING .
  pr INT-MATRIX .
  pr SET{IntVector}
       * (sort NeSet{IntVector} to NeIntVectorSet,
          sort Set{IntVector} to IntVectorSet,
          op _,_ : Set{IntVector} Set{IntVector} -> Set{IntVector}
              to (_,_) [prec 121 format (d d ni d)]) .

  sort IntVectorSetPair .
  op [_|_] : IntVectorSet IntVectorSet -> IntVectorSetPair
             [format (d n++i n ni n-- d)] .
```

Then, the solver is invoked with the built-in operator

```
op natSystemSolve : IntMatrix IntVector String -> IntVectorSetPair
   [special (...)] .
```

which takes as arguments the coefficient matrix, the constant vector, and a string naming the algorithm to be used (see below), and returns the complete set of solutions encoded as a pair of sets of vectors `[ A | B ]`. The non-negative solutions of the linear Diophantine system correspond exactly to those vectors that can be formed as the sum of a vector from A and a non-negative linear combination of vectors from B.

In particular, if the system S is homogeneous (i.e., $c =$ `zeroVector`) then A contains just the constant `zeroVector` and B is the Diophantine basis of S (which will be empty if S only admits the trivial solution). A homogeneous system either has just the trivial solution or infinitely many solutions.

If S is inhomogeneous (i.e., $c \neq$ `zeroVector`) then, if S has no solution, both A and B will be empty; otherwise, B will consist of the Diophantine basis of S', the system formed by setting $c =$ `zeroVector`, while A contains all solutions of S that are not strictly larger than

any element of B. An inhomogeneous system may have no solution (in this case A and B are both empty), a finite number of solutions (in this case A is non-empty and B is empty), or infinitely many solutions (in this case A and B are both non-empty).

In either case, the solution encoding `[ A | B ]` is unique.

Deciding whether a linear Diophantine system admits a non-negative, nontrivial solution is NP-complete (stated as known in [135]). Furthermore the size of the Diophantine basis of a homogeneous system can be very large. For example the equation: $x + y - kz = 0$, for constant $k > 0$, has a Diophantine basis (i.e., set of minimal, nontrivial solutions) of size $k + 1$.

There are currently two algorithms implemented.

The string `"cd"` specifies a version of the classical Contejean-Devie algorithm [37] with various improvements. The algorithm is based on incrementing a vector of counters, one for each variable, and so it can only solve systems where the answers involve fairly small numbers. It is fairly insensitive to the number of degrees of freedom in the problem. The improvements in this implementation take effect when an equation has zero or one unfrozen variables with nonzero coefficients and result in either forced assignments or early pruning of a branch of the search. It performs well on the following homogeneous system from [43],

$$\begin{pmatrix} 1 & 2 & -1 & 0 & -2 & -1 \\ 0 & -1 & -2 & 2 & 0 & 1 \\ 2 & 0 & 1 & -1 & -2 & 0 \end{pmatrix} \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

which has a basis of size 13.

```
Maude> red in DIOPHANTINE :
        natSystemSolve(
          (0,0) |-> 1 ; (0,1) |-> 2 ; (0,2) |-> -1 ;
          (0,3) |-> 0 ; (0,4) |-> -2 ; (0,5) |-> -1 ;
          (1,0) |-> 0 ; (1,1) |-> -1 ; (1,2) |-> -2 ;
          (1,3) |-> 2 ; (1,4) |-> 0 ; (1,5) |-> 1 ;
          (2,0) |-> 2 ; (2,1) |-> 0 ; (2,2) |-> 1 ;
          (2,3) |-> -1 ; (2,4) |-> -2 ; (2,5) |-> 0,
          zeroVector,
          "cd") .
rewrites: 1 in 10ms cpu (46ms real) (100 rews/sec)
result IntVectorSetPair:
  [
    zeroVector
  |
    0 |-> 1 ; 1 |-> 1 ; 4 |-> 1 ; 5 |-> 1,
    0 |-> 1 ; 1 |-> 4 ; 2 |-> 9 ; 3 |-> 11,
    0 |-> 10 ; 1 |-> 4 ; 3 |-> 2 ; 4 |-> 9,
    1 |-> 1 ; 2 |-> 1 ; 3 |-> 1 ; 5 |-> 1,
    1 |-> 8 ; 2 |-> 2 ; 4 |-> 1 ; 5 |-> 12,
    0 |-> 2 ; 1 |-> 4 ; 2 |-> 8 ; 3 |-> 10 ; 4 |-> 1,
    0 |-> 3 ; 1 |-> 4 ; 2 |-> 7 ; 3 |-> 9 ; 4 |-> 2,
    0 |-> 4 ; 1 |-> 4 ; 2 |-> 6 ; 3 |-> 8 ; 4 |-> 3,
    0 |-> 5 ; 1 |-> 4 ; 2 |-> 5 ; 3 |-> 7 ; 4 |-> 4,
    0 |-> 6 ; 1 |-> 4 ; 2 |-> 4 ; 3 |-> 6 ; 4 |-> 5,
    0 |-> 7 ; 1 |-> 4 ; 2 |-> 3 ; 3 |-> 5 ; 4 |-> 6,
    0 |-> 8 ; 1 |-> 4 ; 2 |-> 2 ; 3 |-> 4 ; 4 |-> 7,
    0 |-> 9 ; 1 |-> 4 ; 2 |-> 1 ; 3 |-> 3 ; 4 |-> 8
  ]
```

The string "gcd" specifies an original algorithm based on integer Gaussian elimination followed by a sequence of extended greatest common divisor (gcd) computations. It can "home in" quickly on solutions involving large numbers but it is very sensitive to the number of degrees of freedom and can easily degenerate into a brute force search. Furthermore, termination depends on the bound on the sum of minimal solutions established in [125], which can cause a huge amount of fruitless search after the last minimal solution has been found. It performs well on the "sailors and monkey" problem from [37]:

```
red in DIOPHANTINE :
  natSystemSolve(
    (0,0) |-> 1 ; (0,1) |-> -5 ; (1,1) |-> 4 ; (1,2) |-> -5 ;
    (2,2) |-> 4 ; (2,3) |-> -5 ; (3,3) |-> 4 ; (3,4) |-> -5 ;
    (4,4) |-> 4 ; (4,5) |-> -5 ; (5,5) |-> 4 ; (5,6) |-> -5,
    0 |-> 1 ; 1 |-> 1 ; 2 |-> 1 ; 3 |-> 1 ; 4 |-> 1 ; 5 |-> 1,
    "gcd") .
result IntVectorSetPair:
  [
    0 |-> 15621 ; 1 |-> 3124 ; 2 |-> 2499 ; 3 |-> 1999 ;
    4 |-> 1599 ; 5 |-> 1279 ; 6 |-> 1023
  |
    0 |-> 15625 ; 1 |-> 3125 ; 2 |-> 2500 ; 3 |-> 2000 ;
    4 |-> 1600 ; 5 |-> 1280 ; 6 |-> 1024
  ]
```

Finally, the string "" can be passed as third argument of natSystemSolve, thus allowing the system to choose which algorithm to use. For convenience, the operator

```
op natSystemSolve : IntMatrix IntVector -> IntVectorSetPair .
```

is equationally defined to invoke the built-in operator with ""

```
eq natSystemSolve(M:IntMatrix, V:IntVector)
  = natSystemSolve(M:IntMatrix, V:IntVector, "") .
endfm
```

7.16 Predefined Parameterized Views

Section 7.12 introduces several predefined theories and corresponding default views. Maude's prelude also contains several parameterized views, from TRIV to the corresponding predefined parameterized modules:

```
view List{X :: TRIV}
view WeaklySortableList{X :: STRICT-WEAK-ORDER}
view SortableList{X :: STRICT-TOTAL-ORDER}
view WeaklySortableList'{X :: TOTAL-PREORDER}
view SortableList'{X :: TOTAL-ORDER}
view Set{X :: TRIV}
view List*{X :: TRIV}
view Set*{X :: TRIV}
view Map{X :: TRIV, Y :: TRIV}
view Array{X :: TRIV, Y :: DEFAULT}
```

With these views you can now import module expressions such as LIST{Set{Nat}} and SET{List{String}}. Alternatively, as explained in Section 6.3.7, you can also keep the arguments abstract:

```
fmod LIST-TO-SET-MAP{X :: TRIV, Y :: TRIV} is
  inc MAP{List{X}, Set{Y}} .
endfm
```

Chapter 8

Object-Based Programming

Distributed systems can be naturally modeled in Maude as multisets of entities, loosely coupled by some suitable communication mechanism. An important example is object-based distributed systems in which the entities are objects, each with a unique identity, and the communication mechanism is message passing.

Core Maude supports the modeling of object-based systems by providing a predefined module `CONFIGURATION` that declares sorts representing the essential concepts of object, message, and configuration, along with a notation for object syntax that serves as a common language for specifying object-based systems. In addition, there is an *object-message fair* rewriting strategy that is well suited for executing object system configurations. To specify an object-based system, the user can import `CONFIGURATION` and then define the particular objects, messages, and rules for interaction that are of interest. In addition to simple asynchronous message passing, Maude also supports complex patterns of synchronous interaction that can be used to model higher-level communication abstractions. The user is also free to define his/her own notation for configurations and objects, and can still take advantage of the object-message rewriting strategy, simply by making the appropriate declarations. All this is explained in detail below.

Furthermore, Maude also supports *external objects*, so that objects inside a Maude configuration can interact with different kinds of objects outside it. At present, the external objects directly supported are internet sockets; but through them it is possible to interact with other external objects. In addition, sockets make possible distributed programming with rewrite rules. External objects are discussed in Section 9.

As discussed in Chapter 22, Full Maude provides additional support for object-oriented programming with classes, subclassing, and convenient abbreviations for rule syntax.

8.1 Configurations

The predefined module `CONFIGURATION` in the file `prelude.maude` provides basic sorts and constructors for modeling object-based systems.

```
mod CONFIGURATION is
  *** basic object system sorts
  sorts Object Msg Configuration .

  *** construction of configurations
  subsort Object Msg < Configuration .
  op none : -> Configuration [ctor] .
```

```
op __ : Configuration Configuration -> Configuration
    [ctor config assoc comm id: none] .
```

The basic sorts needed to describe an object system are: `Object`, `Msg` (messages), and `Configuration`. A configuration is a multiset of objects and messages that represents (a snapshot of) a possible system state. Configurations are formed by multiset union (represented by empty syntax, `__`) starting from singleton objects and messages. The empty configuration is represented by the constant `none`. The attribute `config` declares that configurations constructed with `__` support the special object-message fair rewriting behavior (see Section 8.2).

A typical configuration will have the form

⟨*Ob-1*⟩ ... ⟨*Ob-k*⟩ ⟨*Mes-1*⟩ ... ⟨*Mes-n*⟩

where ⟨*Ob-1*⟩, ..., ⟨*Ob-k*⟩ are objects, ⟨*Mes-1*⟩, ..., ⟨*Mes-n*⟩ are messages, and the order is immaterial.

In general, a rewrite rule for an object system has the form

```
rl ⟨Ob-1⟩ ... ⟨Ob-k⟩ ⟨Mes-1⟩ ... ⟨Mes-n⟩
  => ⟨Ob'-1⟩ ... ⟨Ob'-j⟩ ⟨Ob-k+1⟩ ... ⟨Ob-m⟩ ⟨Mes'-1⟩ ... ⟨Mes'-p⟩ .
```

where ⟨*Ob'-1*⟩, ..., ⟨*Ob'-j*⟩ are updated versions of ⟨*Ob-1*⟩, ..., ⟨*Ob-j*⟩ for $j \leq k$, ⟨*Ob-k+1*⟩, ..., ⟨*Ob-m*⟩ are newly created objects, and ⟨*Mes'-1*⟩, ..., ⟨*Mes'-p*⟩ are new messages. An important special case are rules with a single object and at most one message on the lefthand side. These are called *asynchronous* rules. They directly model asynchronous distributed interactions. Rules involving multiple objects are called *synchronous*; they are used to model higher-level communication abstractions.

The user is free to define any object or message syntax that is convenient. However, for uniformity in identifying objects and message receivers, the adopted convention is that *the first argument of an object or message constructor should be an object's name*. This facilitates defining object system rewriting strategies independently of the particular choice of syntax and is essential for using Maude's object-message fair rewriting strategy.

The remainder of the `CONFIGURATION` module provides an object syntax that serves as a common notation that can be used by developers of object-based system specifications. This syntax is also used by Full Maude (see Chapter 22). For this purpose four new sorts are introduced: `Oid` (object identifiers), `Cid` (class identifiers), `Attribute` (a named element of an object's state), and `AttributeSet` (multisets of attributes). Further details about the `CONFIGURATION` module are discussed later in Section 9.

```
  *** Maude object syntax
  sorts Oid Cid .
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet [ctor] .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
      [ctor assoc comm id: none] .
  op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
endm
```

In this syntax, objects have the general form

```
< O : C | ⟨att-1⟩, ..., ⟨att-k⟩ >
```

where `O` is an object identifier, `C` is a class identifier, and ⟨*att-1*⟩, ..., ⟨*att-k*⟩ are the object's attributes. Attribute sets are formed from singleton attributes by a multiset union operator `_,_` with identity `none` (the empty multiset). The `object` attribute in the `<_:_|_>` operator

declares that objects made with this constructor have object-message fair rewriting behavior (see Section 8.2).

Although the user is free to define the syntax of elements of sort `Attribute` according to taste, we will follow the standard Maude notation in most of our examples. The module `BANK-ACCOUNT` illustrates the use of the Maude object syntax to define simple bank account objects. Note that by defining the attribute `bal` with syntax `bal :_` we are able to write account objects as `< A : Account | bal : N >`.

```
mod BANK-ACCOUNT is
  protecting INT .
  inc CONFIGURATION .
  op Account : -> Cid [ctor] .
  op bal :_ : Int -> Attribute [ctor gather (&)] .
  ops credit debit : Oid Nat -> Msg [ctor] .
  vars A B : Oid .
  vars M N N' : Nat .

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

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

The class identifier for bank account objects is `Account`. Each account object has a single attribute named `bal` of sort `Nat` (the account balance). There are two message constructors `credit` and `debit`, each taking an object identifier (the receiver) and a number (the amount to credit or debit). The rule labeled `credit` describes the processing of a credit message and the rule labeled `debit` describes the processing of a debit message. Suppose that constants `A-001`, `A-002`, and `A-003` of sort `Oid` have been declared. Then, the following is an example of a bank account configuration.

```
< A-001 : Account | bal : 300 >
< A-002 : Account | bal : 250 >
< A-003 : Account | bal : 1250 >
debit(A-001, 200)
debit(A-002, 400)
debit(A-003, 300)
credit(A-002, 300)
```

Note that the messages `debit(A-001, 200)` and `debit(A-003, 300)` can be delivered concurrently, either before or after the other messages. However, the message `debit(A-002, 400)` cannot be delivered until after `credit(A-002, 300)` has been delivered, due to the balance condition for the `debit` rule.

The `credit` and `debit` rules are examples of asynchronous message passing rules involving one object and one message on the lefthand side. In these examples no new objects are created and no new messages are sent.

In order to combine the `debit(A-003, 300)` and `credit(A-002, 300)` messages so that the delivery of these two messages becomes a single atomic transaction, we could define a new

message constructor `from_to_transfer_`. The rule for handling a transfer message involves the joint participation of two bank accounts in the transfer, as well as the transfer message. This is an example of a synchronous rule.

```
op from_to_transfer_ : Oid Oid Nat -> Msg [ctor] .
crl [transfer] :
  (from A to B transfer M)
  < A : Account | bal : N >
  < B : Account | bal : N' >
  => < A : Account | bal : N - M >
     < B : Account | bal : N' + M >
  if N >= M .
```

Now we could replace

```
debit(A-003, 300) credit(A-002,300)
```

by

```
from A-003 to A-002 transfer 300
```

in the example configuration. The module `BANK-ACCOUNT-TEST` declares the object identifiers introduced above and defines a configuration constant `bankConf`.

```
mod BANK-ACCOUNT-TEST is
  ex BANK-ACCOUNT .
  ops A-001 A-002 A-003 : -> Oid .
  op bankConf : -> Configuration .
  eq bankConf
    = < A-001 : Account | bal : 300 >
      debit(A-001, 200)
      debit(A-001, 150)
      < A-002 : Account | bal : 250 >
      debit(A-002, 400)
      < A-003 : Account | bal : 1250 >
      (from A-003 to A-002 transfer 300) .
endm
```

From the specification we see that only one of the `debit` messages for `A-001` can be processed. Using the default rewriting strategy we find that the message `debit(A-001, 150)` is processed first in this strategy.

```
Maude> rew in BANK-ACCOUNT-TEST : bankConf .
result Configuration:
  debit(A-001, 200)
  < A-001 : Account | bal : 150 >
  < A-002 : Account | bal : 150 >
  < A-003 : Account | bal : 950 >
```

We use the search command to confirm that it is possible to process the message `debit(A-001, 200)` as well, where the `=>!` symbol indicates that we are searching for states reachable from `bankConf` that cannot be further rewritten (see Sections 5.4.3 and 23.4).

```
Maude> search bankConf =>! C:Configuration debit(A-001, 150) .
search in BANK-ACCOUNT-TEST : bankConf
  =>! C:Configuration debit(A-001, 150) .

Solution 1 (state 8)
```

```
states: 9 rewrites: 49 in 0ms cpu (0ms real) (~ rews/sec)
C:Configuration --> < A-001 : Account | bal : 100 >
        < A-002 : Account | bal : 150 >
        < A-003 : Account | bal : 950 >

No more solutions.
states: 9 rewrites: 49 in 0ms cpu (0ms real) (~ rews/sec)
```

The BANK-MANAGER module below illustrates asynchronous message passing with object creation.

```
mod BANK-MANAGER is
  inc BANK-ACCOUNT .
  op Manager : -> Cid [ctor] .
  op new-account : Oid Oid Nat -> Msg [ctor] .
  vars O C : Oid .
  var  N : Nat .
  rl [new] :
    < O : Manager | none >
    new-account(O, C, N)
    => < O : Manager | none >
       < C : Account | bal : N > .
endm
```

To open a new account, one sends a message to the bank manager with the account name and initial balance, for example, `new-account(A-000, A-004, 100)`. Of course, in a real system more care would be needed to assure unique account identities. To see the bank manager in action, we define the following module.

```
mod BANK-MANAGER-TEST is
  ex BANK-MANAGER .
  ops A-001 A-002 A-003 A-004 : -> Oid .
  op mgrConf : -> Configuration .
  eq mgrConf
    = < A-001 : Account | bal : 300 >
      < A-004 : Manager | none >
      new-account(A-004, A-002, 250)
      new-account(A-004, A-003, 1250) .
endm
```

Then, we rewrite the configuration `mgrConf`:

```
Maude> rew in BANK-MANAGER-TEST : mgrConf .
result Configuration:
  < A-001 : Account | bal : 300 >
  < A-002 : Account | bal : 250 >
  < A-003 : Account | bal : 1250 >
  < A-004 : Manager | none >
```

The relationships between all the modules involved in this example are illustrated in Figure 8.1, where the different types of arrows correspond to the different modes of importation: single arrow for `including`, double arrow for `extending`, and triple arrow for `protecting`.

The examples above illustrate object-based programming in Maude using the common object syntax. Notice that message constructors obey the "first argument is an object identifier" convention. Alternative object syntax is also possible, by defining an associative and commutative configuration constructor and suitable object and message syntax. It is of course also

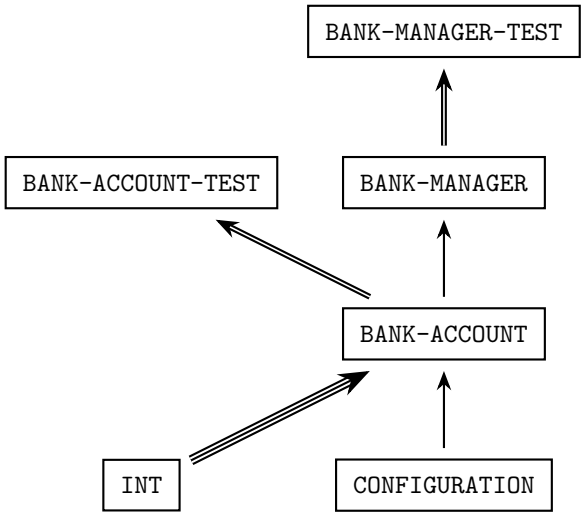


Figure 8.1: Importation graph of bank modules

possible not to use the `config` attribute when defining the multiset union operator, but this will prevent taking advantage of object-message fair rewriting (see Section 8.2). As an example (not using for the moment the `config` attribute, to illustrate different forms of rewriting with objects), we model a *ticker*, the classic example of an actor [1, 134]. First we specify the configurations, objects, and messages of the actor world in the module `ACTOR-CONF`. Actor configurations (of sort `AConf`) are multisets of actors (of sort `Actor`) and messages (of sort `Msg`). Messages are constructed uniformly from an actor identifier and a message body. Thus we introduce sorts `Aid` (actor identifier) and `MsgBody`, and a message constructor `_<|_`.

```
mod ACTOR-CONF is
  sorts Actor Msg AConf .
  subsorts Actor Msg < AConf .
  op none : -> AConf [ctor] .
  op __ : AConf AConf -> AConf [ctor assoc comm id: none] .
  *** actor messages
  sorts Aid MsgBody .
  op _<|_ : Aid MsgBody -> Msg [ctor] .
endm
```

A ticker maintains a counter that it updates in response to a `tick` message. `Ticker(T:Aid, N:Nat)` is an actor with identifier `T:Aid` and counter value `N:Nat`. The ticker sends the current value of its counter in response to a `timeReq` message.

```
mod TICKER is
  including ACTOR-CONF .
  protecting NAT .
  op Ticker : Aid Nat -> Actor [ctor] .
  op tick : -> MsgBody [ctor] .
  op timeReq : Aid -> MsgBody [ctor] .
  op timeReply : Nat -> MsgBody [ctor] .

  vars T C : Aid .
```

```
  var N : Nat .
  rl Ticker(T, N) (T <| tick)
    => Ticker(T, s N) (T <| tick) .
  rl Ticker(T, N) (T <| timeReq(C))
    => Ticker(T, N) (C <| timeReply(N)) .
endm
```

To test the ticker we define actor identifiers for the ticker, `myticker`, a customer, `me`, and an initial configuration with one ticker, one `tick` message, and a `timeReq` message from `me`.

```
mod TICKER-TEST is
  extending TICKER .
  ops myticker me : -> Aid [ctor] .
  op tConf : -> AConf .
  eq tConf
    = Ticker(myticker, 0)
      (myticker <| tick)
      (myticker <| timeReq(me)) .
endm
```

If we ask Maude to rewrite the configuration `tConf` without placing an upper bound on the number of rewrites, Maude will go on forever. This is because there will always be a `tick` message in the configuration, and the ticker can always process this message. Thus we rewrite with an upper bound of, say, 10 rewrites.

```
Maude> rew [10] tConf .
rewrite [10] in TICKER-TEST : tConf myticker <| timeReq(me) .
result AConf:
  (myticker <| tick) (me <| timeReply(1)) Ticker(myticker, 9)
```

We see that the `timeReq` message was processed after just one `tick` was processed.

An interesting property of this configuration is that the reply to the `timeReq` message can contain an arbitrarily large natural number, since any number of `tick`s could be processed before the `timeReq`. For particular numbers this can be checked using the `search` command.

```
Maude> search [1] tConf =>+ tc:AConf me <| timeReply(100) .
search [1] in TICKER-TEST :
  tConf =>+ tc:AConf me <| timeReply(100) .

Solution 1 (state 5152)
states: 5153 rews: 5153 in 0ms cpu (285ms real)(~ rews/sec)

tc:AConf --> (myticker <| tick) Ticker(myticker, 100)
```

Notice that we used the search relation `=>+` (one or more steps) rather than `=>!` (terminating rewrites) since there are no terminal configurations starting from `tConf`. Moreover, we have searched only for the first (`[1]`) solution.

There are two important considerations regarding object systems that are not illustrated by the preceding examples: *uniqueness of object names* and *fairness of message delivery*. To illustrate some of the issues we elaborate the ticker example by defining a ticker factory that creates tickers, and a ticker-customer. The ticker factory accepts requests for new tickers `newReq(c)` where `c` is the customer's name. When such a request is received, a ticker is created and its name is sent to the requesting customer (`newReply(o(a, i))`). To make sure that each ticker has a fresh (unused) name, the ticker factory keeps a counter. It generates ticker names of the form `o(a, i)`, where `a` is the factory name and `i` is the counter value. The counter is

incremented each time a ticker is created. This is just one possible method for assuring unique names for dynamically created objects. If objects are only created by factories that use the above method for generating names, then starting from a configuration of objects with unique names (not of the form `o(a, i)`) the unique name property will be preserved.

```
mod TICKER-FACTORY is
  inc TICKER .
  op TickerFactory : Aid Nat -> Actor [ctor] .
  ops newReq newReply : Aid -> MsgBody [ctor] .
  op o : Aid Nat -> Aid [ctor] .

  vars A C : Aid .
  vars I J : Nat .
  rl [newReq] :
    TickerFactory(A, I) (A <| newReq(C))
    => TickerFactory(A, s I) (C <| newReply(o(A, I)))
       Ticker(o(A, I), 0) (o(A, I) <| tick) .
endm
```

A ticker customer knows the name of a ticker factory. It asks for a ticker, waits for a reply, asks the ticker for the time, waits for a reply, increments its reply counter (used just for the user to monitor customer service) and repeats this process.

```
mod TICKER-CUSTOMER is
  inc TICKER-FACTORY .
  ops Cust Cust1 Cust2 : Aid Aid Nat -> Actor [ctor] .

  vars C TF T : Aid .
  vars N M : Nat .

  rl [req] :
    Cust(C, TF, N)
    => Cust1(C, TF, N) (TF <| newReq(C)) .

  rl [newReply] :
    Cust1(C, TF, N) (C <| newReply(T))
    => Cust2(C, TF, N) (T <| timeReq(C)) .

  rl [timeReply] :
    Cust2(C, TF, N) (C <| timeReply(M))
    => Cust(C, TF, s N) .
endm
```

Now we define a test configuration with a ticker factory and two customers. The importation graph of all the modules involved at this point is shown in Figure 8.2.

```
mod TICKER-FACTORY-TEST is
  ex TICKER-CUSTOMER .
  ops tf c1 c2 : -> Aid [ctor] .
  ops ic1 ic2 : -> AConf .
  eq ic1 = TickerFactory(tf, 0) Cust(c1, tf, 0) .
  eq ic2 = ic1 Cust(c2, tf, 0) .
endm
```

Rewriting this configuration using the `rewrite` command with a bound of 40 results in one ticker being created, and ticking away, while customer `c2` is not given an opportunity to execute at all.

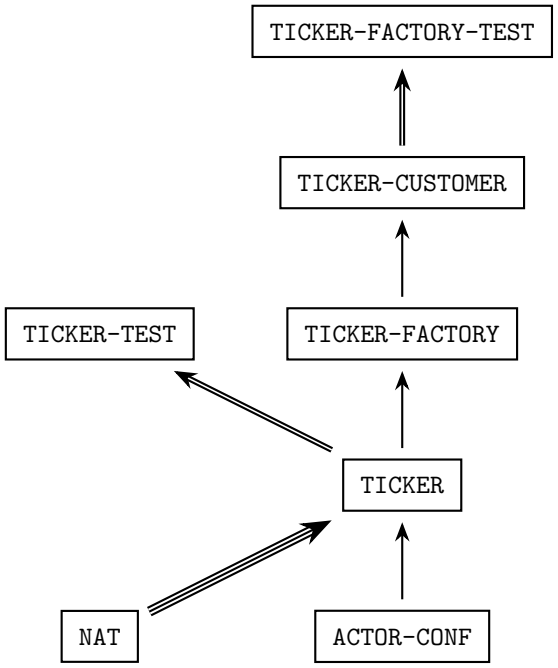


Figure 8.2: Importation graph of ticker modules

```
Maude> rew [40] ic2 .
rewrite [40] in TICKER-FACTORY-TEST : ic2 .
rewrites: 42 in 0ms cpu (0ms real) (~ rewrites/second)
result AConf:
  (o(tf, 0) <| tick)
  Ticker(o(tf, 0), 35) TickerFactory(tf, 1)
  Cust(c1, tf, 1) Cust(c2, tf, 0)
```

In contrast, rewriting using the `frewrite` strategy with the same bound of 40, several tickers are created, however only the first one gets `tick` messages delivered.

```
Maude> frew [40] ic2 .
frewrite [40] in TICKER-FACTORY-TEST : ic2 .
rewrites: 42 in 0ms cpu (1ms real) (~ rewrites/second)
result (sort not calculated):
  (o(tf, 0) <| tick) (o(tf, 1) <| tick)
  (o(tf, 2) <| tick) (o(tf, 3) <| tick)
  (o(tf, 4) <| tick) (o(tf, 5) <| tick)
  (o(tf, 6) <| tick)
  (o(tf, 6) <| timeReq(c1))
  Ticker(o(tf, 0), 6) Ticker(o(tf, 1), 0)
  Ticker(o(tf, 2), 0) Ticker(o(tf, 3), 0)
  Ticker(o(tf, 4), 0) Ticker(o(tf, 5), 0)
  Ticker(o(tf, 6), 0)
  TickerFactory(tf, 7)
  ((tf <| newReq(c2))
  Cust1(c2, tf, 3)) Cust2(c1, tf, 3)
```

The number of rewrites reported by Maude includes both equational and rule rewrites. In the examples above there were 2 equational rewrites (the two equations defining the initial configuration `ic2` and its subconfiguration `ic1`) and 40 rule rewrites. If you execute the command

```
Maude> set profile on .
```

(see Section 20.1.5) before rewriting and then execute

```
Maude> show profile .
```

you will discover that executing the `rewrite` command the rule delivering the `tick` message is used 35 times and the other rules are each used once, while executing the `frewrite` command the `tick` rule is executed only 6 times and each of the other rules are executed between 6 and 8 times.

Turning profiling on substantially reduces performance, so you will want to turn it off

```
Maude> set profile off .
```

when you have found out what you want to know.

Note that `frewrite` uses a fair rewriting strategy, but since it does not know about objects, messages, and configurations, it can only follow a position-fair strategy. As we will explain in the next section, in order to enable the object-message fair rewriting we need only do three things:

- give to the constructor of object and message configurations the `config` attribute,
- give to the message constructor the `message` attribute, and
- give to each object constructor the `object` attribute.

To maintain the separate rewriting semantics we also modify the name of each module by putting an `O` at the front (except for `ACTOR-CONF` which we rename `ACTOR-O-CONF`). Thus we modify the configuration, actor, and message constructor declarations as follows.

```
mod ACTOR-O-CONF is
  ...
  op __ : AConf AConf -> AConf [ctor config assoc comm id: none] .
  op _<|_ : Aid MsgBody -> Msg [ctor message] .
  ...
endm

mod O-TICKER is
  ...
  op Ticker : Aid Nat -> Actor [ctor object] .
  ...
endm

mod O-TICKER-FACTORY is
  ...
  op TickerFactory : Aid Nat -> Actor [ctor object] .
  ...
endm
```

```
mod O-TICKER-CUSTOMER is
  ...
  ops Cust Cust1 Cust2 : Aid Aid Nat -> Actor [ctor object] .
  ...
endm
```

Now the `frewrite` command will use *object-message fair rewriting*, as explained in detail in the next section. The counting of object-message rewrites has two aspects: for the purposes of the rewrite argument given to `frewrite`, a visit to a configuration that results in one or more rewrites counts as a single rewrite; though for other accounting purposes all rewrites are counted. For example, with an upper bound of 40 as above, thirteen tickers are created. To simplify the output we show the results for rewriting with a bound of 20.

```
Maude> frew [20] ic2 .
frewrite [20] in O-TICKER-FACTORY-TEST : ic2 .
rewrites: 76 in 0ms cpu (1ms real) (~ rewrites/second)
result (sort not calculated):
  (o(tf, 0) <| tick) (o(tf, 1) <| tick)
  (o(tf, 2) <| tick) (o(tf, 3) <| tick)
  (o(tf, 4) <| tick) (o(tf, 5) <| tick)
  Ticker(o(tf, 0), 11) Ticker(o(tf, 1), 11)
  Ticker(o(tf, 2), 7) Ticker(o(tf, 3), 7)
  Ticker(o(tf, 4), 3) Ticker(o(tf, 5), 3)
  TickerFactory(tf, 6)
  ((tf <| newReq(c1)) Cust1(c1, tf, 3))
  (tf <| newReq(c2)) Cust1(c2, tf, 3)
```

Notice that each ticker gets a chance to tick (tickers created later will show less time passed), and each customer is treated fairly. In fact using profiling we find that the `tick` rule is used 42 times (which is the total of the counts for the six tickers created), while the other rules are used 6-8 times and there are 2 equational rewrites as before.

Suppose that we try to violate the unique name condition, for example by adding a copy of customer `c1` to the test configuration. When Maude discovers this (it may take a few rewrites), a warning is issued.

```
Maude> frew [4] ic2 Cust(c1, tf, 0) .
Warning: saw duplicate object: Cust1(c1, tf, 0)
frewrite [4] in O-TICKER-FACTORY-TEST : ic2 Cust(c1, tf, 0) .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result AConf:
  (c1 <| newReply(o(tf, 0))) (c1 <| newReply(o(tf, 1)))
  (c2 <| newReply(o(tf, 2)))
  (o(tf, 0) <| tick) (o(tf, 1) <| tick) (o(tf, 2) <| tick)
  Ticker(o(tf, 0), 0) Ticker(o(tf, 1), 0) Ticker(o(tf, 2), 0)
  TickerFactory(tf, 3)
  Cust1(c1, tf, 0) Cust1(c1, tf, 0) Cust1(c2, tf, 0)
```

8.2 Object-message fair rewriting

Object-message fair rewriting is a special rewriting strategy associated with configuration constructors which are declared with the `config` attribute. Configuration constructors must be associative and commutative, and may optionally have an identity element. The empty syntax constructors in the `CONFIGURATION` and `ACTOR-O-CONF` modules above (which have been given

the `config` attribute) are examples of valid configuration constructors, but such default syntax can easily be changed by renaming the `__` operator (see Section 6.2.2). Configurations only have their special behavior with respect to arguments that are constructed using operators that are object or message constructors, that is, they are declared with the `object` or `message` attribute. Such object and message constructors must have at least one argument. Examples include the Maude object constructor in `CONFIGURATION`, the various actor constructors imported into `O-TICKER-FACTORY-TEST`, all of which have been given the `object` attribute, and the actor message constructor which has been given the `message` attribute (which can be abbreviated as `msg`).

An operator can have at most one of the three attributes: `config`, `object`, and `message`. For object constructors, the first argument is considered to be the object's name. For message constructors, the first argument is considered to be the message's target or addressee. There may be multiple configuration, object and message constructors. A rule is considered to be an *object-message rule* if the following requirements hold:

1. Its lefthand side has a configuration constructor on top with two arguments A and B,
2. A and B are stable (that is, they cannot change their top symbol under a substitution),
3. A has a message constructor on top,
4. B has an object constructor on top, and
5. The first arguments of A and B are identical.

For example, the rules `newReply` and `timeReply` in the `O-TICKER-CUSTOMER` module are object-message rules (because configurations are associative and commutative A and B can appear in the rule in either order) while the rule labeled `req` is not, because there is no message term, only an object, in its lefthand side. This rule will be applied in the rewriting that happens after all the enabled object-message rules have been applied, as discussed below.

The object-message fair behavior appears with the command `frewrite` (and at the metalevel with the descent function `metaFrewrite`—see Section 17.6.3). When the fair traversal attempts to perform a single rewrite on a term headed by a configuration constructor, the following happens:

1. Arguments headed by object constructors are collected. It is a runtime error for more than one object to have the same name.
2. For each object, messages with its name as first argument are collected and placed in a queue.
3. Any remaining arguments are placed on a remainder list.
4. For each object, and for each message in its queue, an attempt is made to deliver the message by performing a rewrite with an object-message rule. If all applicable rules fail, the message is placed on the remainder list. If a rule succeeds, the righthand side is constructed, reduced, and the result is searched for the object. Any other arguments in the result configuration go onto the remainder list. If the object cannot be found, any messages left in its queue go onto the remainder list. Once its message queue is exhausted, the object itself is placed on the remainder list.
5. A new term is constructed using the configuration constructor on top of the arguments in the remainder list. This is reduced, and a single rewrite using the non-object-message rules is attempted.

There is no restriction on object names, other than uniqueness. An object may change its object constructor during the course of a rewrite and delivery of any remaining message will still be attempted.[1] If the configuration constructor changes during the course of a rewrite, the resulting term is considered alien, and does not participate any further in the object-message rewriting for the original term. The order in which objects are considered and messages are delivered is system-dependent, but note that newly created messages are not delivered until some future visit to the configuration (though all arguments including new messages and alien configurations could potentially participate in the single non-object-message rewrite attempt). Message delivery is "just" rather than "fair": in order for message delivery to be guaranteed, an object must always be willing to accept the message.[2] If multiple object-message rules contain the same message constructor, they are tried in a round-robin fashion. Non-object-message rules are also tried in a round-robin fashion for the single non-object-message rewrite attempt.

The counting of object-message rewrites is nonstandard: for the purposes of the rewrite argument given to `frewrite`, a visit to a configuration that results in one or more rewrites counts as a single rewrite, though for other accounting purposes all rewrites are counted. Finally, for tracing, profiling, and breakpoints only, there is a fake rewrite at the top of the configuration in the case that object-message rewriting takes place but the single non-object-message rewrite attempt fails. It is not included in the reported rewrite total, but it is inserted to keep tracing consistent.

8.3 Example: data agents

In this section we give an example of a simple distributed dataset in which each agent in a collection of data agents manages a local version of a global data dictionary that maps keys to values. An agent may only have part of the data locally, and must contact other agents to get the value of a key that is not in its local version. To simplify the presentation, we assume that data agents work in pairs.

This example illustrates one way of representing request-reply style of object-based programming in Maude, and also a way of representing information about the state of the task an object is working on when it needs to make one or more requests to other objects in order to answer a request itself. As in the ticker example, we define a uniform syntax for messages. Here, messages have both a receiver and a sender in addition to a message body, and are constructed with the `msg` constructor. The technique for maintaining task information is to define a sort `Request` and a `requests` attribute that holds the set of pending requests. The constant `empty` indicates that an object has no pending request. The request `w4(O:Oid, C:Oid, MB:MsgBody)` indicates that the object is processing a message from `C:Oid` with body `MB:MsgBody` and is waiting for a message from `O:Oid`.

The module `DATA-AGENTS-CONF` extends `CONFIGURATION` with the uniform message syntax and the specification of the sort `Request`.

```
mod DATA-AGENTS-CONF is
  ex CONFIGURATION .
  *** my msg syntax
  sort MsgBody .
  op msg : Oid Oid MsgBody -> Msg [ctor message] .
  *** agents may be pending on requests
```

[1] Assuming, as it should be the case, that both object constructors have been declared with the `object` attribute.

[2] There are program transformations that internalize conditions on message delivery to ensure a stronger fairness condition [101].

```
  sort Request .
  op w4 : Oid Oid MsgBody -> Request [ctor] .
endm
```

A data agent stores a dictionary, mapping keys to data elements. To specify such dictionaries, we use the predefined parameterized module `MAP` (see Section 7.14), renaming the main sort as well as the lookup and update operators as follows:

```
MAP{K, V} * (sort Map{K, V} to Dict{K, V},
             op _[_] to lookup,
             op insert to update) .
```

Remember that the constant `undefined` is the result returned by the lookup operators when the map is not defined on the given key.

We split the specification of data agents into two modules: the parameterized functional module `DATA-AGENTS-INTERFACE`, which defines the interface, and the parameterized system module `DATA-AGENTS` , which gives the rules for agent behavior. This illustrates a technique for modularizing object-based system specifications in order to allow the same interface to be shared by more than one "implementation" (rule set). We already applied this technique in the specification of a vending machine as a system module in Section 5.1. Notice also that `DATA-AGENTS-CONF` is imported in `extending` mode, because we add data to the old sorts, but without making further identifications (the interface module has no equation).

```
mod DATA-AGENTS-INTERFACE{K :: TRIV, V :: TRIV} is
  ex DATA-AGENTS-CONF .

  *** messages
  op getReq : K$Elt -> MsgBody [ctor] .
  op getReply : K$Elt [V$Elt] -> MsgBody [ctor] .
  op setReq : K$Elt V$Elt -> MsgBody [ctor] .
  op setReply : K$Elt [V$Elt] -> MsgBody [ctor] .
  op tellReq : K$Elt V$Elt -> MsgBody [ctor] .
  op tellReply : K$Elt V$Elt -> MsgBody [ctor] .
  op lookupReq : K$Elt -> MsgBody [ctor] .
  op lookupReply : K$Elt [V$Elt] -> MsgBody [ctor] .
endm
```

In a request-reply style of interaction, message body constructors come in pairs. For example, (`lookupReq`, `lookupReply`) and (`tellReq`, `tellReply`) are the message body pairs used when a customer interacts with a data agent in order to access and set data values. Similarly, (`getReq`, `getReply`) and (`setReq`, `setReply`) constitute the message body pairs for an agent to access and set data values from a pal.

A data agent has class identifier `DataAgent`. In addition to the `requests` attribute, each data agent has a `data` attribute holding the agent's local version of the data dictionary, and a `pal` attribute holding the identifier of the other agent. If `sam` and `joe` are collaborating data agents, then their initial state might look like

```
< sam : DataAgent | data : empty, pal : joe, requests : empty >
< joe : DataAgent | data : empty, pal : sam, requests : empty >
```

The module `DATA-AGENTS` specifies a data agent's behavior by giving a rule for handling each type of message it expects to receive (other messages will simply be ignored).

Since we are adding rules acting on the sort `Configuration`, coming from the `CONFIGURATION` module via `DATA-AGENTS-CONF`, we need to make explicit that such modules are imported in `including` mode. We also import in `protecting` mode the predefined parameterized module

SET, instantiated with the following Request view, to define the sets of requests stored in the requests attribute.

```
view Request from TRIV to DATA-AGENTS-CONF is
  sort Elt to Request .
endv

mod DATA-AGENTS{K :: TRIV, V :: TRIV} is
  inc DATA-AGENTS-INTERFACE{K, V} .
  inc DATA-AGENTS-CONF .
  inc CONFIGURATION .
  pr MAP{K, V} * (sort Map{K, V} to Dict{K, V},
                  op _[_] to lookup,
                  op insert to update) .
  pr SET{Request} .

  vars A O C : Oid .
  var  D : Dict{K, V} .
  var  Key : K$Elt .
  vars Val Val' : [V$Elt] .
  var  Atts : AttributeSet .
  var  RS : Set{Request} .

  *** class structure
  op DataAgent : -> Cid [ctor] .
  op data :_ : Dict{K, V} -> Attribute [ctor] .
  op pal :_ : Oid -> Attribute [ctor] .
  op requests :_ : Set{Request} -> Attribute [ctor] .

  *** lookup request
  rl [lookup] :
    < A : DataAgent | data : D, pal : O, requests : RS >
    msg(A, C, lookupReq(Key))
    => if lookup(D, Key) == undefined
       then < A : DataAgent | data : D, pal : O,
                requests : (RS, w4(O, C, lookupReq(Key))) >
            msg(O, A, getReq(Key))
       else < A : DataAgent | data : D, pal : O, requests : RS >
            msg(C, A, lookupReply(Key, lookup(D, Key)))
       fi .

  *** lookup request missing data from pal
  rl [getReq] :
    < A : DataAgent | data : D, pal : O, Atts >
    msg(A, O, getReq(Key))
    => < A : DataAgent | data : D, pal : O, Atts >
       msg(O, A, getReply(Key, lookup(D, Key))) .

  *** receive lookup requested missing data from pal
  rl [getReply] :
    < A : DataAgent | data : D, pal : O,
        requests : (RS, w4(O, C, lookupReq(Key))) >
    msg(A, O, getReply(Key, Val))
    => < A : DataAgent | pal : O, requests : RS,
```

```
          data : if Val == undefined
                 then D
                 else update(Key, Val, D)
                 fi >
       msg(C, A, lookupReply(Key, Val)) .

  *** tell request
  rl [tell] :
    < A : DataAgent | data : D, requests : RS, pal : O >
    msg(A, C, tellReq(Key, Val))
    => if lookup(D, Key) == undefined
       then < A : DataAgent |
              data : D,
              requests : (RS, w4(O, C, tellReq(Key, Val))),
              pal : O >
            msg(O, A, setReq(Key, Val))
       else < A : DataAgent | data : update(Key, Val, D),
              requests : RS, pal : O >
            msg(C, A, tellReply(Key, Val))
       fi .
  *** request update for missing data from pal
  rl [setReq] :
    < A : DataAgent | data : D, pal : O, Atts >
    msg(A, O, setReq(Key, Val))
    => if lookup(D, Key) == undefined
       then < A : DataAgent | data : D, pal : O, Atts >
            msg(O, A, setReply(Key, undefined))
       else < A : DataAgent | data : update(Key, Val, D),
              pal : O, Atts >
            msg(O, A, setReply(Key, Val))
       fi .

  *** receive requested update for missing data from pal
  rl [setReply] :
    < A : DataAgent | data : D, pal : O,
        requests : (RS, w4(O, C, tellReq(Key, Val))) >
    msg(A, O, setReply(Key, Val'))
    => < A : DataAgent | pal : O, requests : RS,
          data : if Val' == undefined
                 then update(Key, Val, D)
                 else D
                 fi >
       msg(C, A, tellReply(Key, Val)) .
endm
```

The rule labeled `lookup` specifies how an agent handles a `lookupReq` message. The agent first looks to see if its local dictionary contains the requested entry. If `lookup(D, Key) == undefined`, then a `getReq` is sent to the `pal` and the agent waits for a reply, remembering the pending lookup request (`w4(O, C, lookupReq(Key))`). If the agent has the requested entry, then it is returned in a `lookupReply` message.

The rules labeled `getReq` and `getReply` specify how agents exchange dictionary entries. An agent can always answer a `getReq` message, since the `Atts` variable will match any status attribute. The agent simply replies with the result, possibly `undefined`, of looking up the requested key in its local dictionary. An agent only expects a `getReply` message if it has made

a request, and this only happens if the agent is trying to handle a `lookupReq` message. Thus the rule only matches if the agent has the appropriate request `w4(O, C, lookupReq(Key))` in its `requests` attribute. The agent records the received reply with `update(Key, Val, D)` when this reply is not `undefined`, and in any case sends it on to the customer with the message `msg(C, A, lookupReply(Key, Val))`.

The rules labeled `tell`, `setReq`, and `setReply` specify how an agent handles a `tellReq` message, following a protocol similar to the one described for the lookup request.

Note that in the case of agents with just these three attributes, using the `Atts` variable of sort `AttributeSet` or the `requests : RS` expression, with `RS` a variable of sort `Set{Request}`, are equivalent ways of saying that the rule matches any set of requests. The first way is more extensible, in that the rule would still work for agents belonging to a subclass of `DataAgent` that uses additional attributes.

To test the data agent specification, we define a module `AGENT-TEST`. This module defines object identifiers `sam` and `joe` for data agents, and `me` to name an external customer. It also defines an initial configuration containing two agents named `sam` and `joe` with empty data dictionaries, and two initial `tellReq` messages for each agent. We take both keys and data elements to be quoted identifiers, by instantiating the parameterized `DATA-AGENTS` module with the predefined `Qid` view.

```
mod AGENT-TEST is
  ex DATA-AGENTS{Qid, Qid} .
  ops sam joe me : -> Oid [ctor] .
  op iconf : -> Configuration .
  eq iconf
    = < sam : DataAgent | data : empty,
          pal : joe, requests : empty >
      msg(sam, me, tellReq('a, 'bc))
      msg(sam, me, tellReq('d, 'ef))
      < joe : DataAgent | data : empty,
          pal : sam, requests : empty >
      msg(joe, me, tellReq('g, 'hi))
      msg(joe, me, tellReq('j, 'kl)) .
endm
```

The importation graph of all the modules involved in this example is shown in Figure 8.3, where the three different types of arrows correspond to the three different modes of importation.

The following are results from test runs. First we rewrite the initial configuration `iconf`, resulting in a configuration in which the agents have updated appropriately their data, and there is one reply for each `tellReq` message.

```
Maude> rew iconf .
result Configuration:
  < sam : DataAgent | data : ('a |-> 'bc, 'd |-> 'ef),
      pal : joe, requests : empty >
  < joe : DataAgent | data : ('g |-> 'hi, 'j |-> 'kl),
      pal : sam, requests : empty >
  msg(me, sam, tellReply('a, 'bc))
  msg(me, sam, tellReply('d, 'ef))
  msg(me, joe, tellReply('g, 'hi))
  msg(me, joe, tellReply('j, 'kl))
```

Next we try adding a lookup request and discover that, using Maude's default rewriting strategy, the lookup request is delivered before the tell requests, so the reply is `undefined`.

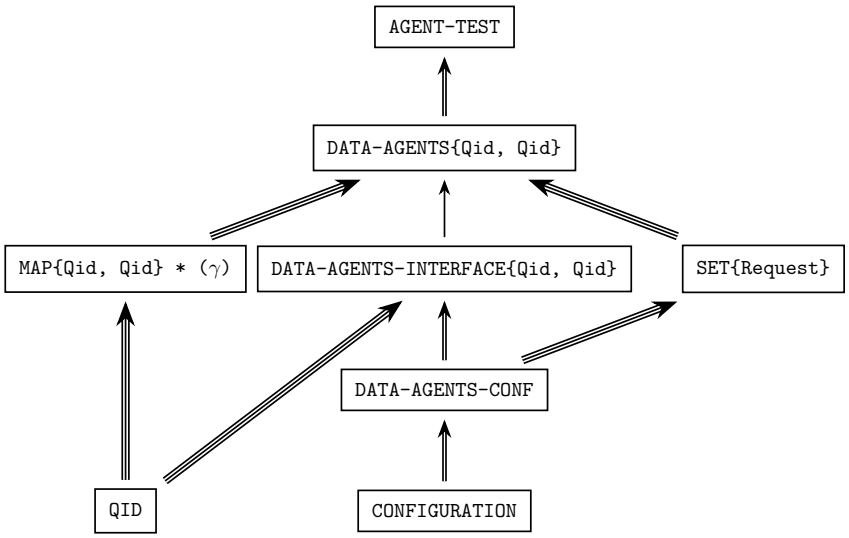


Figure 8.3: Importation graph of data-agents modules

```
Maude> rew iconf msg(sam, me, lookupReq('a)) .
result Configuration:
  < sam : DataAgent | data : ('a |-> 'bc, 'd |-> 'ef),
      pal : joe, requests : empty >
  < joe : DataAgent | data : ('g |-> 'hi, 'j |-> 'kl),
      pal : sam, requests : empty >
  msg(me, sam, tellReply('a, 'bc))
  msg(me, sam, tellReply('d, 'ef))
  msg(me, sam, lookupReply('a, undefined))
  msg(me, joe, tellReply('g, 'hi))
  msg(me, joe, tellReply('j, 'kl))
```

To see if a good answer can be obtained, we use the `search` command to look for a state in which there is a `lookupReply` with data entry different from `undefined`.

```
Maude> search iconf msg(sam, me, lookupReq('a))
        =>! msg(me, sam, lookupReply('a, Q:Qid)) C:Configuration .

Solution 1 (state 1081)
C:Configuration -->
  < sam : DataAgent | data : ('a |-> 'bc, 'd |-> 'ef),
      pal : joe, requests : empty >
  < joe : DataAgent | data : ('g |-> 'hi, 'j |-> 'kl),
      pal : sam, requests : empty >
  msg(me, sam, tellReply('a, 'bc))
  msg(me, sam, tellReply('d, 'ef))
  msg(me, joe, tellReply('g, 'hi))
  msg(me, joe, tellReply('j, 'kl))
Q:Qid --> 'bc

No more solutions.
```

Indeed, there is just one such reply.

Notice that two collaborating agents may get inconsistent data, that is, different values for the same key, if they receive simultaneously tell requests for the same key. We may use the `search` command to illustrate how this may happen.

```
Maude> search iconf
             msg(sam, me, tellReq('m, 'no))
             msg(joe, me, tellReq('m, 'pq))
         =>! C:Configuration
             < sam : DataAgent |
                 data : ('m |-> Q:Qid, R:Dict{Qid, Qid}),
                 Atts:AttributeSet >
             < joe : DataAgent |
                 data : ('m |-> Q':Qid, R':Dict{Qid, Qid}),
                 Atts':AttributeSet >
          such that Q:Qid =/= Q':Qid .

Solution 1 (state 5117)
C:Configuration -->
   msg(me, sam, tellReply('a, 'bc))
   msg(me, sam, tellReply('d, 'ef))
   msg(me, sam, tellReply('m, 'no))
   msg(me, joe, tellReply('g, 'hi))
   msg(me, joe, tellReply('j, 'kl))
   msg(me, joe, tellReply('m, 'pq))
Atts:AttributeSet --> pal : joe, requests : empty
R:Dict{Qid,Qid} --> 'a |-> 'bc, 'd |-> 'ef
Q:Qid --> 'no
Atts':AttributeSet --> pal : sam, requests : empty
R':Dict{Qid,Qid} --> 'g |-> 'hi, 'j |-> 'kl
Q':Qid --> 'pq

No more solutions.
```

Note the use of the `such that` condition to filter `search` solutions (see Section 5.4.3 and 23.4).

Chapter 9

External Objects and IO

We use the object-message system to allow interactions with external objects that represent external entities with state. This section explains Maude's support for rewriting with external objects and an implementation of sockets, standard streams, files, processes, and meta-interpreters as the first such external objects.

Configurations that want to communicate with external objects must contain at least one *portal*, where

```
sort Portal .
subsort Portal < Configuration .
op <> : -> Portal [ctor] .
```

is part of the predefined module `CONFIGURATION` in the file `prelude.maude`. From an implementation point of view, the main purpose of having a portal term in a configuration is to avoid the degenerate case of a configuration that consists just of an object waiting for a message from outside of the configuration. This would be problematic because the special behavior for object-message rewriting and exchanging messages with external objects is attached to the configuration constructor.

Rewriting with external objects is started by the external rewrite command `erewrite` (abbreviated `erew`) which is like `frewrite` (see Sections 5.4 and 8.2) except that it allows messages to be exchanged with external objects that do not reside in the configuration. Currently the command `erewrite` has some severe limitations, which might be fixed in future releases:

1. Maude will check for external events after every fair traversal of the term. When there are no internal rewrites available but there are requests pending on external objects (such as reading on a socket or waiting for a process to exit), rather than finishing and returning to the command line, Maude will suspend on external events. An external event may result in a message injected into a configuration that enables more internal rewrites.

2. Rewrites that involve messages entering or leaving the configuration do not show up in tracing, profiling, or rewrite counts.

Note that, even if there are no more rewrites possible, `erewrite` may not terminate; if there are requests made to external objects that have not yet been fulfilled because of waiting for external events from the operating system, the Maude interpreter will suspend until at least one of those events occurs, at which time rewriting will resume. While the interpreter is suspended, the command `erewrite` may be aborted with `^C`. External objects created by an `erewrite` command do not survive to the next `erewrite`. If a message to an external object is ill-formed

or inappropriate, or the external object is not ready for it, it just stays in the configuration for future acceptance or for debugging purposes.

Certain predefined external objects are available and some of them are object managers that can create more ephemeral external objects that represent entities such as files and sockets, and as we will see in Chapter 19, virtual copies of the Maude interpreter itself.

Access to external objects represents a potential threat. It is difficult for a user to tell just by looking at it, if an arbitrary Maude program contains malware. Furthermore, a Maude program can assemble meta-malware using innocent looking code and then execute it in a meta-interpreter. File handling and process execution are now disabled by default. These features need to be enabled with the command line flags `-allow-files` and `-allow-processes`. To run completely trusted code, the command line flag `-trust` enables all dangerous features. See Section 23.1.

9.1 Standard streams

Each Unix process has three I/O channels, called *standard streams*: standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). In Maude, these are represented as three unique external objects, that are defined in a predefined module `STD-STREAM` in the `file.maude` file included in the Maude distribution. Because some of the messages that are useful for streams are also useful for file I/O, these messages are pulled out into a module `COMMON-MESSAGES`.

```
mod COMMON-MESSAGES is
  protecting STRING .
  including CONFIGURATION .

  op gotLine : Oid Oid String -> Msg [ctor msg format (m o)] .
  op write : Oid Oid String -> Msg [ctor msg format (b o)] .
  op wrote : Oid Oid -> Msg [ctor msg format (m o)] .
endm

mod STD-STREAM is
  including COMMON-MESSAGES .

  op getLine : Oid Oid String -> Msg [ctor msg ...] .
  op stdin : -> Oid [special (...)] .
  op stdout : -> Oid [special (...)] .
  op stderr : -> Oid [special (...)] .
endm
```

I/O on standard streams is always line-oriented and in text mode. `stdout` and `stderr` accept the `write()` message just like file handlers. They are automatically flushed so this is the only message they can accept. They always return a `wrote()` message.

The `stdin` object accepts a three-argument `getLine()` message:

```
getLine(stdin, ME, PROMPT)
```

`ME` is the identifier of the sender object, to which the answer will be sent. `PROMPT` is a string that will be shown to inform the user that some input is expected. Once the message is processed, when the user hits the return key, `stdin` replies with a message

```
gotLine(ME, stdin, TEXT)
```

where `TEXT` contains just a `"\n"` character in the case of an empty line, and is empty in the case of an error or end-of-file (`^D`).

9.1.1 The Hello Word! example

The module HELLO below shows a very simple program implementing an interaction with the user, which is asked to introduce his/her name to be properly greeted. The equation for run produces a starting configuration, containing the portal, a user object to receive messages, and a message to stdin to read a line of text from the keyboard. When stdin has a line of text, it sends the text to the requesting object in a gotLine message.

```
mod HELLO is
  including STD-STREAM .

  op myClass : -> Cid .
  op myObj : -> Oid .
  op run : -> Configuration .

  var O : Oid .
  var A : AttributeSet .
  var S : String .
  var C : Char .

  eq run
   = <>
     < myObj : myClass | none >
     getLine(stdin, myObj, "What is your name? ") .

  rl < myObj : myClass | A >
     gotLine(myObj, O, S)
  => < myObj : myClass | A >
     if S =/= ""
     then write(stdout, myObj, "Hello " + S)
     else none
     fi .
endm

Maude> erew run .
What is your name? Joe
Hello Joe
result Configuration: <> wrote(myObj, stdout) < myObj : myClass | none >
```

9.1.2 A ROT13 cypher example

As a second example of the use of standard streams, let us consider the specification in the module ROT13 below. The example prints out a banner and then reads lines from the keyboard, encrypts each line using the ROT13 cypher, and finally prints out the result.

```
mod ROT13 is
  including STD-STREAM .
  protecting INT .

  op myClass : -> Cid .
  op myObj : -> Oid .
  op run : -> Configuration .
  op rot13 : String -> String .
```

```
  vars O O2 : Oid .
  var  A : AttributeSet .
  vars S T : String .
  var  C : Char .

  eq rot13(C)
   = if C >= "A" and C <= "Z"
     then char(ascii("A") + ((13 + (ascii(C) - ascii("A"))) rem 26))
     else if C >= "a" and C <= "z"
          then char(ascii("a") + ((13 + (ascii(C) - ascii("a"))) rem 26))
          else C
          fi
     fi .
  eq rot13(S)
   = rot13(substr(S, 0, 1)) + rot13(substr(S, 1, length(S)))
   [owise] .

  eq run
   = <>
     < myObj : myClass | none >
     write(stdout, myObj, "\nROT13 Encryption\n----------------\n") .

  rl < O : myClass | A >
     wrote(O, O2)
  => < O : myClass | A >
     getLine(stdin, O, "Enter plain text> ") .
  rl < O : myClass | A >
     gotLine(O, O2, T)
  => < O : myClass | A >
     if T =/= ""
     then write(stdout, O, "Cypher text: " + rot13(T) + "\n")
     else none
     fi .
endm
```

The equation for `run` produces a starting configuration, containing the portal, a user object to receive messages, and an initial message to `stdout` to print out the banner. Each time `stdout` prints out some text, it sends a `wrote` reply to the requesting object; this is matched in the first rule which then sends a `getLine` message to `stdin` to read a line of text from the keyboard. When `stdin` has a line of text, it sends the text to the requesting object in a `gotLine` message; the second rule matches this message, calls the equationally defined `rot13` function to encrypt the text, and sends a message to `stdout` to print the encrypted text. When the `wrote` reply comes back, the sequence repeats.

We can have the following interaction:

```
Maude> erewrite run .
erewrite in ROT13 : run .

ROT13 Encryption
----------------
Enter plain text> Maude
Cypher text: Znhqr
Enter plain text>
rewrites: 107 in 0ms cpu (3364ms real) (149025 rewrites/second)
```

```
result Configuration: <> < myObj : myClass | none >
```

9.1.3 A calculator example

The following program illustrates the use of functions `metaReduce` and `metaParse` (from module `META-LEVEL`, described later in Chapter 17) and functions `tokenize` and `printTokens` (from module `LEXICAL` in Section 7.11) to read Maude arithmetic expressions from the terminal and evaluate them.

```
mod CALCULATOR is
  including STD-STREAM .
  protecting LEXICAL .
  protecting META-LEVEL .

  op myClass : -> Cid .
  op myObj : -> Oid .
  op run : -> Configuration .

  vars O O2 : Oid .
  var  A : AttributeSet .
  vars S T : String .

  op compute : String -> String .
  eq compute(S) = compute2(metaParse(['CONVERSION], tokenize(S), anyType)) .

  op compute2 : ResultPair? -> String .
  eq compute2({T:Term, Q:Qid}) = compute3(metaReduce(['CONVERSION], T:Term)) .
  eq compute2(noParse(N:Nat)) = printTokens('\r) + "syntax error" + printTokens('\o) .

  op compute3 : ResultPair? -> String .
  eq compute3({T:Term, Q:Qid}) = printTokens(metaPrettyPrint(['CONVERSION], T:Term)) .

  eq run
   = <>
     < myObj : myClass | none >
     write(stdout, myObj, "\nCalculator\n------------\n") .

  rl < O : myClass | A >
     wrote(O, O2)
  => < O : myClass | A >
     getLine(stdin, O, "Expression> ") .
  rl < O : myClass | A >
     gotLine(O, O2, T)
  => < O : myClass | A >
     if T == ""
     then none
     else write(stdout, O, "Answer: " + compute(T) + "\n")
     fi .
endm
```

Then we can execute, for example:

```
Maude> erew run .
erewrite in CALCULATOR : run .
```

```
Calculator
------------
Expression> 3/6 + 12/32
Answer: 7/8
Expression>
```

9.2 File I/O

As pointed out in the introduction of the chapter, access to external objects represents a potential threat. File handling is disabled by default. The feature may be enabled with the command line flags `-allow-files` or `-trust`. See Section 23.1.

Unlike standard streams, of which there are exactly three, a Unix process may have many different files open at any one time. Thus, in order to create new file handle objects as needed, we have a unique external object called `fileManager`. To open a file, the `fileManager` is sent a message `openFile`. On success, an `openedFile` message is returned, with the name of an external object that is a handle on the open file as one of its arguments. Messages to read and write the file can be directed to the handle object. On failure, a `fileError` message is returned, with a text explanation of why the file could not be opened as one of its arguments. These messages are defined in the module `FILE`, which is distributed as part of the Maude system in the `file.maude` file.

```
mod FILE is
  including COMMON-MESSAGES .
  protecting INT .

  sort Base .
  ops start current end : -> Base [ctor] .

  op file : Nat -> Oid [ctor] .

  op openFile : Oid Oid String String -> Msg [ctor msg format (b o)] .
  op openedFile : Oid Oid Oid -> Msg [ctor msg format (m o)] .

  op getLine : Oid Oid -> Msg [ctor msg format (b o)] .

  op getChars : Oid Oid Nat -> Msg [ctor msg format (b o)] .
  op gotChars : Oid Oid String -> Msg [ctor msg format (m o)] .

  op flush : Oid Oid -> Msg [ctor msg format (b o)] .
  op flushed : Oid Oid -> Msg [ctor msg format (m o)] .

  op setPosition : Oid Oid Int Base -> Msg [ctor msg format (b o)] .
  op positionSet : Oid Oid -> Msg [ctor msg format (m o)] .
  op getPosition : Oid Oid -> Msg [ctor msg format (b o)] .
  op positionGot : Oid Oid Nat -> Msg [ctor msg format (m o)] .

  op closeFile : Oid Oid -> Msg [ctor msg format (b o)] .
  op closedFile : Oid Oid -> Msg [ctor msg format (m o)] .

  op fileError : Oid Oid String -> Msg [ctor msg format (m o)] .
```

```
  op fileManager : -> Oid [special (...)] .
endm
```

This API basically wraps the C *stdio* library. To open a file you send `fileManager` a message

```
openFile(fileManager, ME, PATH, MODE)
```

where `ME` is the name of the object the reply should be sent to, `PATH` is the path of the file you want to open, and `MODE` is one of: `r`, `r+`, `w`, `w+`, `a`, `a+`.[1]

The reply is either

```
openedFile(ME, fileManager, FILE-HANDLE)
```

or

```
fileError(ME, fileManager, REASON)
```

where `FILE-HANDLE` is the name of the object used to access the file, and `REASON` is a textual explanation from the operating system of what went wrong.

If the file was opened for writing you can send

```
write(FILE-HANDLE, ME, DATA)
```

to write data to the file, and receive a reply

```
wrote(ME, FILE-HANDLE)
```

or

```
fileError(ME, FILE-HANDLE, REASON)
```

A file that is opened for writing can also be sent

```
flush(FILE-HANDLE, ME)
```

to flush any buffered data and receive a reply

```
flushed(ME, FILE-HANDLE)
```

or

```
fileError(ME, FILE-HANDLE, REASON)
```

A file that is opened for reading can be read on a line-by-line basis using the message:

```
getLine(FILE-HANDLE, ME)
```

where the reply is either

```
gotLine(ME, FILE-HANDLE, TEXT)
```

[1] The opening modes are exactly the same as those for the C standard library function `fopen()`. The following text in provided by the BSD `fopen` manpage (see `https://www.freebsd.org/`):

> The argument mode points to a string beginning with one of the following letters:
>
> **"r"** Open for reading. The stream is positioned at the beginning of the file. Fail if the file does not exist.
>
> **"w"** Open for writing. The stream is positioned at the beginning of the file. Truncate the file to zero length if it exists or create the file if it does not exist.
>
> **"a"** Open for writing. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file [...]. Create the file if it does not exist.
>
> An optional "+" following "r", "w", or "a" opens the file for both reading and writing.

or

```
fileError(ME, FILE-HANDLE, REASON)
```

Here TEXT includes the newline character (if present, since end-of-file also ends the current line). TEXT is empty to indicate end-of-file with no more characters to read.

A file that is opened for reading can also be read on a character basis (which is more appropriate for binary files) using the message

```
getChars(FILE-HANDLE, ME, #CHARS-TO-GET)
```

where the reply is either

```
gotChars(ME, FILE-HANDLE, DATA)
```

or

```
fileError(ME, FILE-HANDLE, REASON)
```

Here if DATA is shorter than requested, it indicates the end-of-file was reached.

Reading and writing share a common position into the file where the next read or write takes place. This position in bytes from the start of the file can be obtained by sending the message

```
getPosition(FILE-HANDLE, ME)
```

where the reply is either

```
positionGot(ME, FILE-HANDLE, OFFSET)
```

or

```
fileError(ME, FILE-HANDLE, REASON)
```

The position may be changed with the message

```
setPosition(FILE-HANDLE, ME, OFFSET, BASE)
```

Here the OFFSET is relative to BASE, where BASE can take one of three values: start, the start of the file, current, the current position, or end, the end of the file. In the current and end cases, negative values of OFFSET are allowed. The reply is either

```
positionSet(ME, FILE-HANDLE)
```

or

```
fileError(ME, FILE-HANDLE, REASON)
```

Finally, an open file can be closed with the message

```
closeFile(FILE-HANDLE, ME)
```

Since it is always OK to close an open file, the response is always

```
closedFile(ME, FILE-HANDLE)
```

Note that messages that are not recognized or that are sent to nonexistent objects will be silently ignored and left in the configuration. Messages that are recognized but are not appropriate for the object they are sent to or which have bad arguments will similarly be ignored but will generate a “message declined” advisory.

9.2.1 A file copy example

The COPY-FILE module below illustrates the basic use of files. It specifies a simple algorithm to copy files. In this case, the `run` operator takes two arguments, namely the name of the file to be copied and the name of the new file. The equation for `run` produces a starting configuration, containing the portal, a user object to receive messages, and an initial message to open the original file. Once it is opened, the new file is created. Notice the `"w"` argument of the `openFile` message. Once both files are opened, a loop in which a line is read from the original file and written in the copy file is initiated. This loop ends when the end of the file is reached. Both files are then closed.

```
view Oid from TRIV to CONFIGURATION is
  sort Elt to Oid .
endv

fmod MAYBE{X :: TRIV} is
  sort Maybe{X} .
  subsort X$Elt < Maybe{X} .
  op maybe : -> Maybe{X} [ctor] .
endfm

mod COPY-FILE is
  including FILE .
  protecting (MAYBE * (op maybe to null)){Oid} .

  op myClass : -> Cid .
  op myObj : -> Oid .
  ops in:_ out:_ : Maybe{Oid} -> Attribute .
  ops inFile:_ outFile:_ : String -> Attribute .

  op run : String String -> Configuration .
  vars Text Original Copy : String .
  vars FHIn FHOut : Oid .
  var  Attrs : AttributeSet .

  eq run(Original, Copy)
   = <>
     < myObj : myClass | in: null, inFile: Original,
         out: null, outFile: Copy >
     openFile(fileManager, myObj, Original, "r") .

  rl < myObj : myClass | in: null, outFile: Copy, Attrs >
     openedFile(myObj, fileManager, FHIn)
  => < myObj : myClass | in: FHIn, outFile: Copy, Attrs >
     openFile(fileManager, myObj, Copy, "w") .
  rl < myObj : myClass | in: FHIn, out: null, Attrs >
     openedFile(myObj, fileManager, FHOut)
  => < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     getLine(FHIn, myObj) .
  rl < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     gotLine(myObj, FHIn, Text)
  => < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     if Text == ""
     then closeFile(FHIn, myObj)
```

```
        closeFile(FHOut, myObj)
     else write(FHOut, myObj, Text)
     fi .
  rl < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     wrote(myObj, FHOut)
  => < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     getLine(FHIn, myObj) .
  rl < myObj : myClass | in: FHIn, out: FHOut, Attrs >
     closedFile(myObj, FHIn)
     closedFile(myObj, FHOut)
  => none .
endm
```

You can then execute the program to copy a file `"in.txt"` with the following command:

```
Maude> erew run("in.txt", "out.txt") .
```

9.3 Sockets

The sockets external objects are accessed using the messages declared in the following `SOCKET` module, included in the file `socket.maude` which is part of the Maude distribution.

```
mod SOCKET is
  protecting STRING .
  including CONFIGURATION .

  op socket : Nat -> Oid [ctor] .

  op createClientTcpSocket : Oid Oid String Nat -> Msg
       [ctor msg format (b o)] .
  op createServerTcpSocket : Oid Oid Nat Nat -> Msg
       [ctor msg format (b o)] .
  op createdSocket : Oid Oid Oid -> Msg [ctor msg format (m o)] .

  op acceptClient : Oid Oid -> Msg [ctor msg format (b o)] .
  op acceptedClient : Oid Oid String Oid -> Msg
       [ctor msg format (m o)] .

  op send : Oid Oid String -> Msg [ctor msg format (b o)] .
  op sent : Oid Oid -> Msg [ctor msg format (m o)] .

  op receive : Oid Oid -> Msg [ctor msg format (b o)] .
  op received : Oid Oid String -> Msg [ctor msg format (m o)] .

  op closeSocket : Oid Oid -> Msg [ctor msg format (b o)] .
  op closedSocket : Oid Oid String -> Msg [ctor msg format (m o)] .

  op socketError : Oid Oid String -> Msg [ctor msg format (r o)] .

  op socketManager : -> Oid [special (...)] .
endm
```

Currently only IPv4 TCP sockets are supported; other protocol families and socket types may be added in the future. The external object named by the constant `socketManager` is a factory for socket objects.

To create a client socket, you send `socketManager` a message

```
createClientTcpSocket(socketManager, ME, ADDRESS, PORT)
```

where `ME` is the name of the object the reply should be sent to, `ADDRESS` is the name of the server you want to connect to (say "www.google.com"), and `PORT` is the port you want to connect to (say 80 for HTTP connections). You may also specify the name of the server as an IPv4 dotted address or as "localhost" for the same machine where the Maude system is running on.

The reply will be either

```
createdSocket(ME, socketManager, NEW-SOCKET-NAME)
```

or

```
socketError(ME, socketManager, REASON)
```

where `NEW-SOCKET-NAME` is the name of the newly created socket and `REASON` is the operating system's terse explanation of what went wrong.

You can then send data to the server with a message

```
send(SOCKET-NAME, ME, DATA)
```

which elicits either

```
sent(ME, SOCKET-NAME)
```

or

```
closedSocket(ME, SOCKET-NAME, REASON)
```

Notice that all errors on a client socket are handled by closing the socket.

Similarly, you can receive data from the server with a message

```
receive(SOCKET-NAME, ME)
```

which elicits either

```
received(ME, SOCKET-NAME, DATA)
```

or

```
closedSocket(ME, SOCKET-NAME, REASON)
```

When you are done with the socket, you can close it with a message

```
closeSocket(SOCKET-NAME, ME)
```

with reply

```
closedSocket(ME, SOCKET-NAME, "")
```

Once a socket has been closed, its name may be reused, so sending messages to a closed socket can cause confusion and should be avoided.

Notice that TCP does not preserve message boundaries, so sending `"one"` and `"two"` might be received as `"on"` and `"etwo"`. Delimiting message boundaries is the responsibility of the next higher-level protocol, such as HTTP. We will present an implementation of buffered sockets in Section 9.3.2 which solves this problem.

9.3.1 An HTTP/1.0 client example

The following modules implement an updated version of the five rule HTTP/1.0 client from the paper "Towards Maude 2.0" [27] that is now executable. The first module defines some auxiliary operations on strings.

```
fmod STRING-OPS is
  protecting STRING .
  var S : String .
  op extractHostName : String -> String .
  op extractPath : String -> String .
  op extractHeader : String -> String .
  op extractBody : String -> String .

  eq extractHostName(S)
    = if find(S, "/", 0) == notFound
      then S
      else substr(S, 0, find(S, "/", 0))
      fi .

  eq extractPath(S)
    = if find(S, "/", 0) == notFound
      then "/"
      else substr(S, find(S, "/", 0), length(S))
      fi .

  eq extractHeader(S)
    = substr(S, 0, find(S, "\r\n\r\n", 0) + 4) .
  eq extractBody(S)
    = substr(S, find(S, "\r\n\r\n", 0) + 4, length(S)) .
endfm
```

The second module requests one web page from an HTTP server.

```
mod HTTP/1.0-CLIENT is
  protecting STRING-OPS .
  including SOCKET .
  sort State .
  ops idle connecting sending receiving closing : -> State [ctor] .
  op state:_ : State -> Attribute [ctor] .
  op requester:_ : Oid -> Attribute [ctor] .
  op url:_ : String -> Attribute [ctor] .
  op stored:_ : String -> Attribute [ctor] .

  op HttpClient : -> Cid .
  op httpClient : -> Oid .
  op dummy : -> Oid .

  op getPage : Oid Oid String -> Msg [msg ctor] .
  op gotPage : Oid Oid String String -> Msg [msg ctor] .

  vars H R R' TS : Oid .
  vars U S ST : String .
```

First, we try to connect to the server using port 80, updating the state and the `requester` attribute with the new server.

```
rl [getPage] :
  getPage(H, R, U)
  < H : HttpClient |
      state: idle, requester: R', url: S, stored: "" >
  => < H : HttpClient |
         state: connecting, requester: R, url: U, stored: "" >
     createClientTcpSocket(socketManager, H,
       extractHostName(U), 80) .
```

Once we are connected to the server (we have received a `createdSocket` message), we send a `GET` message (from the HTTP protocol) requesting the page. When the message is sent, we wait for a response.

```
rl [createdSocket] :
  createdSocket(H, socketManager, TS)
  < H : HttpClient |
      state: connecting, requester: R, url: U, stored: "" >
  => < H : HttpClient |
         state: sending, requester: R, url: U, stored: "" >
     send(TS, H, "GET " + extractPath(U) + " HTTP/1.0\r\nHost: " +
     extractHostName(U) + "\r\n\r\n") .

rl [sent] :
  sent(H, TS)
  < H : HttpClient |
      state: sending, requester: R, url: U, stored: "" >
  => < H : HttpClient |
         state: receiving, requester: R, url: U, stored: "" >
     receive(TS, H) .
```

While the page is not complete, we receive data and append it to the string on the `stored` attribute. When the page is completed, the server closes the socket, and then we show the page information by means of the `gotPage` message.

```
rl [received] :
  received(H, TS, S)
  < H : HttpClient |
      state: receiving, requester: R, url: U, stored: ST >
  => receive(TS, H)
     < H : HttpClient | state: receiving,
         requester: R, url: U, stored: (ST + S) > .

rl [closedSocket] :
  closedSocket(H, TS, S)
  < H : HttpClient |
      state: receiving, requester: R, url: U, stored: ST >
  => gotPage(R, H, extractHeader(ST), extractBody(ST)) .
```

We use a special operator `start` to represent the initial configuration. It receives the server URL we want to connect to. Notice the occurrence of the portal `<>` in such initial configuration.

```
op start : String -> Configuration .
eq start(S)
  = <>
    getPage(httpClient, dummy, S)
    < httpClient : HttpClient | state: idle, requester: dummy,
```

```
        url: "", stored: "" > .
endm
```

Now we can get pages from servers, say "www.google.com", by using the following Maude command (note the ellipsis in the output):

```
Maude> erew start("www.google.com") .
result Configuration: <> gotPage(dummy, httpClient, "HTTP/1.0 ... </html>")
```

It is also possible to have optional bounds on the `erewrite` command, and then use the continuation commands to get more results, like, for example,

```
Maude> erew [1, 2] start("www.google.com") .
result Configuration:
  <>
  < httpClient : HttpClient |
      state: connecting,
      requester: dummy,
      url: "www.google.com",
      stored: "" >
  createClientTcpSocket(socketManager, httpClient, "www.google.com", 80)

Maude> cont 1 .
result Configuration:
  <>
  < httpClient : HttpClient |
      state: connecting,
      requester: dummy,
      url: "www.google.com",
      stored: "" >
```

To have communication between two Maude interpreter instances, one of them must take the server role and offer a service on a given port; generally ports below 1024 are protected. You cannot in general assume that a given port is available for use. To create a server socket, you send `socketManager` a message

```
createServerTcpSocket(socketManager, ME, PORT, BACKLOG)
```

where `PORT` is the port number and `BACKLOG` is the number of queue requests for connection that you will allow (5 seems to be a good choice). The response is either

```
createdSocket(ME, socketManager, SERVER-SOCKET-NAME)
```

or

```
socketError(ME, socketManager, REASON)
```

Here `SERVER-SOCKET-NAME` refers to a server socket. The only thing you can do with a server socket (other than close it) is to accept clients, by means of the following message:

```
acceptClient(SERVER-SOCKET-NAME, ME)
```

which elicits either

```
acceptedClient(ME, SERVER-SOCKET-NAME, ADDRESS, NEW-SOCKET-NAME)
```

or

```
socketError(ME, socketManager, REASON)
```

Here ADDRESS is the originating address of the client and NEW-SOCKET-NAME is the name of the socket you use to communicate with that client. This new socket behaves just like a client socket for sending and receiving. Note that an error in accepting a client does not close the server socket. You can always reuse the server socket to accept new clients until you explicitly close it.

The following modules illustrate a very naive two-way communication between two Maude interpreter instances. The issues of port availability and message boundaries are deliberately ignored for the sake of illustration (and thus if you are unlucky this example could fail).

The first module describes the behavior of the server.

```
mod FACTORIAL-SERVER is
  inc SOCKET .
  pr CONVERSION .
  op _! : Nat -> NzNat .
  eq 0 ! = 1 .
  eq (s N) ! = (s N) * (N !) .

  op Server : -> Cid .
  op aServer : -> Oid .

  vars O LISTENER CLIENT : Oid .
  var  A : AttributeSet .
  var  N : Nat .
  vars IP DATA S : String .
```

Using the following rules, the server waits for clients. If one client is accepted, the server waits for messages from it. When the message arrives, the server converts the received data to a natural number, computes its factorial, converts it into a string, and finally sends this string to the client. Once the message is sent, the server closes the socket with the client.

```
  rl [createdSocket] :
    < O : Server | A > createdSocket(O, socketManager, LISTENER)
    => < O : Server | A > acceptClient(LISTENER, O) .

  rl [acceptedClient] :
    < O : Server | A > acceptedClient(O, LISTENER, IP, CLIENT)
    => < O : Server | A > receive(CLIENT, O)
       acceptClient(LISTENER, O) .

  rl [received] :
    < O : Server | A > received(O, CLIENT, DATA)
    => < O : Server | A >
       send(CLIENT, O, string(rat(DATA, 10)!, 10)) .

  rl [sent] :
    < O : Server | A > sent(O, CLIENT)
    => < O : Server | A > closeSocket(CLIENT, O) .

  rl [closedSocket] :
    < O : Server | A > closedSocket(O, CLIENT, S)
    => < O : Server | A > .
endm
```

The Maude command that initializes the server is as follows, where the configuration includes the portal <>.

```
Maude> erew <>
           < aServer : Server | none >
           createServerTcpSocket(socketManager, aServer, 8811, 5) .
```

The second module describes the behavior of the clients.

```
mod FACTORIAL-CLIENT is
  inc SOCKET .
  op Client : -> Cid .
  op aClient : -> Oid .

  vars O CLIENT : Oid .
  var  A : AttributeSet .
```

Using the following rules, the client connects to the server (clients must be created after the server), sends a message representing a number,[2] and then waits for the response. When the response arrives, there are no blocking messages and rewriting ends.

```
  rl [createdSocket] :
    < O : Client | A > createdSocket(O, socketManager, CLIENT)
    => < O : Client | A > send(CLIENT, O, "6") .

  rl [sent] :
    < O : Client | A > sent(O, CLIENT)
    => < O : Client | A > receive(CLIENT, O) .
endm
```

The initial configuration for the client will be as follows, again with portal `<>`.

```
Maude> erew <>
           < aClient : Client | none >
           createClientTcpSocket(socketManager,
             aClient, "localhost", 8811) .
```

Almost everything in the socket implementation is done in a nonblocking way; so, for example, if you try to open a connection to some webserver and that webserver takes 5 minutes to respond, other rewriting and transactions happen in the meanwhile as part of the same command `erewrite`. The one exception is DNS resolution, which is done as part of the `createClientTcpSocket` message handling and which cannot be nonblocking without special tricks.

9.3.2 Buffered sockets

As we said before, TCP does not preserve message boundaries; to guarantee it we may use a filter class `BufferedSocket`, defined in the module `BUFFERED-SOCKET`, which is described here. We interact with buffered sockets in the same way we interact with sockets, with the only difference that all messages in the module `SOCKET` have been capitalized to avoid confusion. Thus, to create a client with a buffered socket, you send `socketManager` a message

```
CreateClientTcpSocket(socketManager, ME, ADDRESS, PORT)
```

instead of a message

```
createClientTcpSocket(socketManager, ME, ADDRESS, PORT).
```

[2] In this quite simple example, it is always `"6"`.

All the messages have exactly the same declarations, the only difference being their initial capitalization:

```
op CreateClientTcpSocket : Oid Oid String Nat -> Msg
     [ctor msg format (b o)] .
op CreateServerTcpSocket : Oid Oid Nat Nat -> Msg
     [ctor msg format (b o)] .
op CreatedSocket : Oid Oid Oid -> Msg [ctor msg format (m o)] .

op AcceptClient : Oid Oid -> Msg [ctor msg format (b o)] .
op AcceptedClient : Oid Oid String Oid -> Msg
     [ctor msg format (m o)] .

op Send : Oid Oid String -> Msg [ctor msg format (b o)] .
op Sent : Oid Oid -> Msg [ctor msg format (m o)] .

op Receive : Oid Oid -> Msg [ctor msg format (b o)] .
op Received : Oid Oid String -> Msg [ctor msg format (m o)] .

op CloseSocket : Oid Oid -> Msg [ctor msg format (b o)] .
op ClosedSocket : Oid Oid String -> Msg [ctor msg format (m o)] .

op SocketError : Oid Oid String -> Msg [ctor msg format (r o)] .
```

Thus, apart from this small difference, we interact with buffered sockets in exactly the same way we do with sockets, the boundary control being completely transparent to the user.

When a buffered socket is created, in addition to the socket object through which the information will be sent, a `BufferedSocket` object is also created on each side of the socket (one in each one of the configurations between which the communication is established). All messages sent through a buffered socket are manipulated before they are sent through the socket underneath. When a message is sent through a buffered socket, a mark is placed at the end of it; the `BufferedSocket` object at the other side of the socket stores all messages received on a buffer, in such a way that when a message is requested the marks placed indicate which part of the information received must be given as the next message.

An object of class `BufferedSocket` has two attributes: `read`, of sort `String`, which stores the messages read, and `bState`, which indicates whether the filter is `idle` or `active`.

```
op BufferedSocket : -> Cid [ctor] .
op read :_ : String -> Attribute [ctor gather(&)] .
op bState :_ : BState -> Attribute [ctor gather(&)] .
sort BState .
ops idle active : -> BState [ctor] .
```

The identifiers of the `BufferedSocket` objects are marked with a `b` operator, i.e., the buffers associated with a socket `SOCKET` have identifier `b(SOCKET)`. Note that there is a `BufferedSocket` object on each side of the socket, that is, there are two objects with the same identifier, but in different configurations.

```
op b : Oid -> Oid [ctor] .
```

A buffered socket object understands capitalized versions of the messages a socket object understands. For most of them, it just converts them into the corresponding uncapitalized message. There are messages `AcceptClient`, `CloseSocket`, `CreateServerTcpSocket`, and `CreateClientTcpSocket` with the same arities as the corresponding socket messages, with the following rules.

```
vars SOCKET NEW-SOCKET SOCKET-MANAGER O : Oid .
vars ADDRESS IP IP' DATA S S' REASON : String .
var  Atts : AttributeSet .
vars PORT BACKLOG N : Nat .

rl [createServerTcpSocket] :
  CreateServerTcpSocket(SOCKET-MANAGER, O, PORT, BACKLOG)
  => createServerTcpSocket(SOCKET-MANAGER, O, PORT, BACKLOG) .

rl [acceptClient] :
  AcceptClient(SOCKET, O)
  => acceptClient(SOCKET, O) .

rl [closeSocket] :
  CloseSocket(b(SOCKET), SOCKET-MANAGER)
  => closeSocket(SOCKET, SOCKET-MANAGER) .

rl [createClientTcpSocket] :
  CreateClientTcpSocket(SOCKET-MANAGER, O, ADDRESS, PORT)
  => createClientTcpSocket(SOCKET-MANAGER, O, ADDRESS, PORT) .
```

Note that in these cases the buffered-socket versions of the messages are just translated into the corresponding socket messages.

A `BufferedSocket` object can also convert an uncapitalized message into the capitalized one. The rule `socketError` shows this:

```
rl [socketError] :
  socketError(O, SOCKET-MANAGER, REASON)
  => SocketError(O, SOCKET-MANAGER, REASON) .
```

`BufferedSocket` objects are created and destroyed when the corresponding sockets are. Thus, we have rules

```
rl [acceptedclient] :
  acceptedClient(O, SOCKET, IP', NEW-SOCKET)
  => AcceptedClient(O, b(SOCKET), IP', b(NEW-SOCKET))
     < b(NEW-SOCKET) : BufferedSocket |
         bState : idle, read : "" > .

rl [createdSocket] :
  createdSocket(O, SOCKET-MANAGER, SOCKET)
  => < b(SOCKET) : BufferedSocket | bState : idle, read : "" >
     CreatedSocket(O, SOCKET-MANAGER, b(SOCKET)) .

rl [closedSocket] :
  < b(SOCKET) : BufferedSocket | Atts >
  closedSocket(SOCKET, SOCKET-MANAGER, DATA)
  => ClosedSocket(b(SOCKET), SOCKET-MANAGER, DATA) .
```

Once a connection has been established, and a `BufferedSocket` object has been created on each side, messages can be sent and received. When a `Send` message is received, the buffered socket sends a `send` message with the same data plus a mark[3] to indicate the end of the message.

[3]We use the character '`#`' as mark; therefore, the user data sent through the sockets should not contain such a character.

```
rl [send] :
  < b(SOCKET) : BufferedSocket | bState : active, Atts >
  Send(b(SOCKET), O, DATA)
  => < b(SOCKET) : BufferedSocket | bState : active, Atts >
     send(SOCKET, O, DATA + "#") .

rl [sent] :
  < b(SOCKET) : BufferedSocket | bState : active, Atts >
  sent(O, SOCKET)
  => < b(SOCKET) : BufferedSocket | bState : active, Atts >
     Sent(O, b(SOCKET)) .
```

The key is then in the reception of messages. A `BufferedSocket` object is always listening to the socket. It sends a `receive` message at start up and puts all the received messages in its buffer. Notice that a buffered socket goes from `idle` to `active` in the `buffer-start-up` rule. A `Receive` message is then handled if there is a complete message in the buffer, that is, if there is a mark on it, and results in the reception of the first message in the buffer, which is removed from it.

```
rl [buffer-start-up] :
  < b(SOCKET) : BufferedSocket | bState : idle, Atts >
  => < b(SOCKET) : BufferedSocket | bState : active, Atts >
     receive(SOCKET, b(SOCKET)) .

rl [received] :
  < b(SOCKET) : BufferedSocket |
      bState : active, read : S, Atts >
  received(b(SOCKET), O, DATA)
  => < b(SOCKET) : BufferedSocket |
         bState : active, read : (S + DATA), Atts >
     receive(SOCKET, b(SOCKET)) .

crl [Received] :
  < b(SOCKET) : BufferedSocket |
      bState : active, read : S, Atts >
  Receive(b(SOCKET), O)
  => < b(SOCKET) : BufferedSocket |
         bState : active, read : S', Atts >
     Received(O, b(SOCKET), DATA)
  if N := find(S, "#", 0)
     /\ DATA := substr(S, 0, N)
     /\ S' := substr(S, N + 1, length(S)) .
```

The `BUFFERED-SOCKET` module is used in the specification of Mobile Maude, a mobile agent language based on Maude, which is discussed in detail in [29, Chapter 16].

9.4 Processes

Unix processes can be handled as Maude external objects through the API contained in the `process.maude` file. As pointed out in the introduction of the chapter, access to external objects represents a potential threat. Process execution is disabled by default. The feature may be enabled with the command line flags `-allow-processes` or `-trust`. See Section 23.1.

The external object `processManager` allows us to create new processes (with the `createProcess` message), with which we may directly interact.

```
fmod STRING-LIST is
  protecting LIST{String} *
    (sort NeList{String} to NeStringList, sort List{String} to StringList) .
endfm

mod PROCESS is
  including SOCKET .
  protecting STRING-LIST .

  sorts ProcessOption ProcessOptionSet .
  subsort ProcessOption < ProcessOptionSet .
  op none : -> ProcessOptionSet [ctor] .

  sort ExitStatus .
  op normalExit : Nat -> ExitStatus [ctor] .
  op terminatedBySignal : String -> ExitStatus [ctor] .

  op process : Nat -> Oid [ctor] .

  op createProcess : Oid Oid String StringList ProcessOptionSet -> Msg [ctor msg format (b o)] .
  op createdProcess : Oid Oid Oid Oid Oid -> Msg [ctor msg format (m o)] .

  op signalProcess : Oid Oid String -> Msg [ctor msg format (b o)] .
  op signaledProcess : Oid Oid -> Msg [ctor msg format (m o)] .

  op waitForExit : Oid Oid -> Msg [ctor msg format (b o)] .
  op exited : Oid Oid ExitStatus -> Msg [ctor msg format (m o)] .

  op processError : Oid Oid String -> Msg [ctor msg format (r o)] .

  op processManager : -> Oid [special (...)] .
endm
```

New processes are created with the `createProcess` message, which has five arguments:

- As usual, the first argument is the addressee of the message, in this case the special builtin object `processManager`, a constant of sort `Oid`;
- the second argument is the `Oid` of the requesting object, to which the response will be sent;
- the third argument is a `String` with the system command that executes the new process;
- the fourth argument is a `StringList` with the arguments to the command; and
- the fifth argument is a `ProcessOptionSet`, which is reserved for future use (currently we will use `none`).

On success, the reply to a `createProcess` message is a message `createdProcess`, which also carries five arguments, all of type `Oid`:

- the requesting object,
- the sender object (the `processManager` object),
- the identifier of the new process object, which uses the `process` constructor,

- the identifier of the new socket object for stdin/stdout, and
- the identifier of the new socket object for stderr.

One can communicate with the new process using its stdin/stdout socket and the normal socket interface defined in `socket.maude` (see Section 9.3). Error messages can be read on the stderr socket. It is probably good practice to always have receives pending on both sockets so the process is not stalled due to socket buffers filling. The stderr socket ignores `send` messages.

Maude will stop rewriting and return to the command line as soon as there are no rewrites possible and no external events pending that could change the rewrite state, even if the newly created process is still running. It is therefore good practice to wait for a process to exit, if only to avoid filling the process table with zombies. This may be done by sending the process a `waitForExit` message. If such a message is sent, Maude will not return to the command line until the process actually exits, which is what happens when there are no further pending external events. When the process finally exits, the reply message is `exited`, with the last argument being either `normalExit` with an 8-bit exit code for a normal exit or a `terminatedBySignal` with the name of the signal in the case of termination by signal. The `waitForExit` message is nonblocking, so it can be sent any time after the `createdProcess` message is received. A second `waitForExit` message to the same process will not be accepted.

When a process has been waited for, its stdin/stdout and stderr are closed. If processes are not waited to be closed, it is the user's responsibility to close the sockets to avoid leaking file descriptors. One way to do this is to keep receiving any final characters from the process until an end-of-file or error condition arises and Maude returns a `closedSocket` message. Alternatively, sockets can be closed by explicitly sending them `closeSocket` messages.

Different executables have different conventions for telling them to quit. You can send a process the empty string to cause an EOF condition on its stdin (see below). You can also send a signal to a process using the `signalProcess` message, whose third argument is a string with the name of the Unix signal.[4] The process responds with a `signaledProcess` message.

The message `processError` is used as the reply for errors coming from the operating system. In particular a `processError` reply is generated if the child process cannot execute the specified executable.

We provide simple examples illustrating these features in the following sections.

9.4.1 A desk calculator process

Our first example invokes the dc (desk calculator) program to do some calculations and then kills it with a `SIGTERM` signal. To interact with the external objects we use an object of class `User` with attributes `state` (to keep track of the state the object is in), process (to keep the `Oid` of the external object), `io` and `err` (to keep the socket objects' identifiers), and `result` (to store the response). Please, see inlined comments explaining the rules.

```
mod PROCESS-DC is
  inc PROCESS .
  inc MAYBE{Oid} * ( op maybe to null ).
  op me : -> Oid .
  op User : -> Cid .
  op state:_ : Nat -> Attribute .
  ops process:_ io:_ err:_ : Maybe{Oid} -> Attribute .
```

[4] Not every signal is supported by every platform and not every signal makes sense to send to a child process. The following POSIX signals are supported on all Unix platforms: `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGKILL`, `SIGALRM`, `SIGTERM`, `SIGSTOP`, and `SIGCONT`; everything else should be considered platform dependent.

```
  op result:_ : String -> Attribute .

  vars X P ERR IO : Oid .
  vars Result Str : String .
  var  Atts : AttributeSet .

  ---- once the process is created, a request to the dc and a waitForExit are sent
  rl < X : User | state: 1, process: null, io: null, err: null, Atts >
     createdProcess(X, processManager, P, IO, ERR)
  => < X : User | state: 2, process: P, io: IO, err: ERR, Atts >
     send(IO, X, "10 16 + p\n") ---- dc keeps the result in p
     waitForExit(P, X) .
  ---- once the message has been sent, a response is requested
  rl < X : User | state: 2, io: IO, Atts >
     sent(X, IO)
  => < X : User | state: 3, io: IO, Atts >
     receive(IO, X) .
  ---- upon reception of the response, a new request is sent
  ---- the useful part of the response is stored in the result attribute
  rl < X : User | state: 3, io: IO, result: "", Atts >
     received(X, IO, Result)                        ---- Result --> "26\n"
  => < X : User | state: 4, io: IO,
       result: substr(Result, 0, sd(length(Result), 1)), Atts >
     send(IO, X, "4 * p\n") .
  ---- once the message has been sent, a new response is requested
  rl < X : User | state: 4, io: IO, Atts >
     sent(X, IO)
  => < X : User | state: 5, io: IO, Atts >
     receive(IO, X) .
  ---- once this response is received, the process is terminated with a SIGTERM signal
  rl < X : User | state: 5, process: P, io: IO, result: Str, Atts >
     received(X, IO, Result)                        ---- Result --> "104\n"
  => < X : User | state: 6, process: P, io: IO,
       result: substr(Result, 0, sd(length(Result), 1)), Atts >
     signalProcess(P, X, "SIGTERM") .
  ---- signaledProcess and exited messages are received to confirm the operation
  rl < X : User | state: 6, process: P, Atts >
     signaledProcess(X, P)
     exited(X, P, terminatedBySignal("SIGTERM"))
  => < X : User | state: 7, process: P, Atts > .
endm
```

We can run the above program with the following `erewrite` command:

```
Maude> erew <>
          < me : User | state: 1, process: null, io: null, err: null, result: "" >
          createProcess(processManager, me, "dc", nil, none) .
result Configuration:
  <>
  < me : User | state: 7,
                process: process(66941), io: socket(3), err: socket(5),
                result: "104" >
```

The execution terminates with the `User` object in its final state, with `"104"` as final result of the operations carried out by the dc process. Notice that when the current `erewrite` context

is destroyed (say, by running a new rewrite command), Maude will kill any processes and close any sockets that correspond to Maude external objects in that context.

9.4.2 Python and Maude processes

Each application has its own form of interaction. In this section we present a Maude program that interacts with a second Maude process, and, to analyze its output, it uses a Python program that is run on the Python 3 interpreter. Specifically, the program runs an external Maude process in which it loads the `process-dc.maude` file presented in the previous section. As shown there, the program uses the dc to do a calculation. Since the output is rather complex, we use a Python program to process the given output and extract the part we are interested in. The output is processed in a Python interpreter process using the function `select_text`. The string `"104"` is returned as result.

Although just for illustration purposes in this case, the use of other languages applications may be convenient in some of our programs. In this case, the use of regular expressions for the manipulation of strings comes quite handy. Our Python function `select_text` is very simple; it uses the functionality provided in the `re` module for searching a string `s`, passed as first argument, for a regular expression with the exact pattern we are interested in, which is given as second argument `p`:

```
def select_text(s, p):
    result = re.compile(p).search(s)
    if result:
        return result.group(1)
    else:
        raise Exception("not␣found")
```

As for the example in Section 9.4.1, we use an object to keep track of the process and information of the different sockets. Since we are going to have two of these processes, we can introduce the appropriate definitions as a generic class.

```
mod PROCESS-PROXY is
  inc PROCESS .
  inc MAYBE{Oid} * ( op maybe to null ) .

  ---- class Proxy | state: Nat, process: Maybe{Oid}, io: Maybe{Oid}, err: Maybe{Oid} .
  sort Proxy .
  subsort Proxy < Cid .
  op Proxy : -> Proxy .
  op state:_ : Nat -> Attribute .
  ops process:_ io:_ err:_ : Maybe{Oid} -> Attribute . ---- process and channels
endm
```

In this program, we assume that both the Maude system (`maude.darwin64`) and the Python interpreter (`python3`) are in your path, and that the `process-dc.maude` file from the previous section is in your working directory. If this is not the case, please change the code below appropriately. A `PATH` constant is given for the location of Maude's executable, which should be changed depending on your setting.

The functionality provided by the `select_text` Python function may be useful in different contexts. To make it available as an auxiliary function, we show below the `PROCESS-PYTHON` module, which provides a function `string-extract` that extracts a substring from a given string that matches a given pattern. Specifically, given a `string-extract(X, Y, Str, Pat)` message, it returns either `string-extracted(Y, X, Result)`, with the matched substring, or `string-extract-error(Y, X, Result)`, if any error happens.

```
mod PROCESS-PYTHON is
  inc PROCESS-PROXY .

  ---- class Python | string: String, pattern: String, requester: Maybe{Oid} .
  ---- subclass Python < Proxy .
  ops string:_ pattern:_ : String -> Attribute [ctor] .
  op requester:_ : Maybe{Oid} -> Attribute [ctor] .
  sort Python .
  subsort Python < Proxy .
  op Python : -> Python .

  op string-extract : Oid Oid String String -> Msg .
  ops string-extracted string-extract-error : Oid Oid String -> Msg .

  vars X Y P ERR IO : Oid .
  vars Result Str Pat : String .
  var  Atts : AttributeSet .

  rl string-extract(X, Y, Str, Pat)
  => < X : Python | state: 1, process: null, io: null, err: null,
                    requester: Y, string: Str, pattern: Pat >
     createProcess(processManager, X, "python3", "-iu" "string_extract.py", none) .
  rl < X : Python | state: 1, process: null, io: null, err: null,
                    string: Str, pattern: Pat, Atts >
     createdProcess(X, processManager, P, IO, ERR)
  => < X : Python | state: 2, process: P, io: IO, err: ERR, Atts >
     send(IO, X, "select_text(’’’" + Str + "’’’, ’" + Pat + "’)\n")
     waitForExit(P, X) .
  rl < X : Python | state: 2, io: IO, err: ERR, Atts >
     sent(X, IO)
  => < X : Python | state: 3, io: IO, err: ERR, Atts >
     receive(IO, X)
     receive(ERR, X) .
  rl < X : Python | state: 3, io: IO, requester: Y, Atts > ---- output received
     received(X, IO, Result)                                ---- Result --> "’104’\n"
  => < X : Python | state: 4, io: IO, requester: Y, Atts >
     string-extracted(Y, X, substr(Result, 1, sd(length(Result), 3)))
     send(IO, X, "quit()\n") .
  rl < X : Python | state: 4, io: IO, Atts >              ---- quit sent
     sent(X, IO)
  => < X : Python | state: 5, io: IO, Atts > .
  rl < X : Python | err: ERR, requester: Y, Atts >        ---- error
     received(X, ERR, Result)
  => < X : Python | err: ERR, requester: Y, Atts >        ---- error msg might be empty
     if Result == ">>> ... ... ... >>> " then none
     else string-extract-error(Y, X, Result) fi .
  rl < X : Python | state: 5, process: P, Atts >          ---- process terminated
     exited(X, P, normalExit(0))
  => none .
endm
```

The rules are explained in inline comments, but several issues are worth pointing out:

- Although `string-extract` is declared as a message, notice that its first argument is the identifier used to create the proxy.

- The string and the pattern need to be kept by the wrapper until the reception of the `createdProcess` message.
- The Python process is started in interactive mode (using the `-i` flag), and the file `string_extract.py` with the above `select_text` function is also passed as a command argument.
- In Python, strings may use either single quotes (`'`), quotes (`"`), or triple quotes (`"""` or `'''`). Since the output from the Maude process may contain quotes and new line characters, the string to search in is passed using triple single quotes. The pattern is passed using single quotes, assuming that it does not contain new line characters. Quotes could also be used in both cases, but then the string on which the search is performed should have to be manipulated before it is sent as an argument of the Python command. Remember that quotes inside strings must be double quoted, i.e., a string like `"\"hi\""` must be sent as argument of the function as `"select_text(\"\\\"hi\\\"\")"`. Notice that, since to put a quote inside a string you use `\"`, and to put a backslash inside a string you use `\\`, then, to put `\"` inside a string you must use `\\\"`.
- The process is terminated using its quit command. Once the process is terminated, the wrapper is removed.

The following module uses the above `string-extract` function to extract the result from the execution of the `process-dc.maude` program.

Although the rules are explained in inlined comments below, there are a couple of issues worth pointing out:

- The `process-dc.maude` file is passed as an argument of the `maude.darwin64` command when the `createProcess` is sent. Notice also that Maude has to be invoked with the `-trust` flag to run external processes.
- Maude gives a welcome header when started, and when a command is sent to it, it first acknowledges the command being executed, and then gives the actual output when it finishes.
- In this case, we use a constant `init` to initiate the execution. The output and potential error messages are given with respective operators `result` and `error`.

```
mod PROCESS-MAUDE is
  ---- Maude's path, empty string if in the system path or in the same directory
  op PATH : -> String .
  eq PATH = "" . ---- E.g., "/Users/duran/maude/maude-3.1"

  inc PROCESS-PYTHON .

  ops result error : String -> Msg .

  op init : -> Configuration .
  ops me python : -> Oid .

  vars X Y P ERR IO : Oid .
  vars Result Str Pat : String .
  var  Atts : AttributeSet .
```

```
---- the program begins by creating the Maude process
rl init
=> < me : Proxy | state: 1, process: null, io: null, err: null >
   createProcess(processManager, me,
     PATH + "maude.darwin64", "-trust" "process-dc.maude", none) .
---- Upon creation of the Maude process, it requests its banner
rl < X : Proxy | state: 1, process: null, io: null, err: null, Atts >
   createdProcess(X, processManager, P, IO, ERR)
=> < X : Proxy | state: 2, process: P, io: IO, err: ERR, Atts >
   receive(IO, X)
   waitForExit(P, X) .
---- When the banner is received, the command is sent
---- This is the same command used in the previous section
rl < X : Proxy | state: 2, io: IO, Atts >
   received(X, IO, Result)                          ---- Maude's header
=> < X : Proxy | state: 3, io: IO, Atts >
   send(IO, X,
     "erew <> " +
        "< me : User | state: 1, process: null, io: null, err: null, result: \"\" >"
        + " createProcess(processManager, me, \"dc\", nil, none) .\n") .
---- When the sending is confirmed, a request for input is sent
rl < X : Proxy | state: 3, io: IO, Atts >
   sent(X, IO)
=> < X : Proxy | state: 4, io: IO, Atts >
   receive(IO, X) .
---- Maude first sends the command being executed
---- In this case we send a receive message on both channels
rl < X : Proxy | state: 4, io: IO, err: ERR, Atts >
   received(X, IO, Result)              ---- command being executed
=> < X : Proxy | state: 5, io: IO, err: ERR, Atts >
   receive(IO, X)     ---- request output
   receive(ERR, X) .  ---- and potential error message
---- Once an answer is received, a quit command is sent to the Maude
---- process to terminate it, and a string-extract message is delivered
rl < X : Proxy | state: 5, io: IO, Atts > ---- result from ok execution
   received(X, IO, Result)
=> < X : Proxy | state: 6, io: IO, Atts >
   send(IO, X, "q\n")
   string-extract(python, X, Result, "result: \"([^\"]*)\"") .
---- the response from the string-extract message is handled
rl < X : Proxy | state: 6, io: IO, Atts > ---- result from ok execution
   string-extracted(X, Y, Result)
=> < X : Proxy | state: 7, io: IO, Atts > ---- result from ok execution
   result(Result) .
rl < X : Proxy | Atts >                ---- error execution
   string-extract-error(X, Y, Result)
=> < X : Proxy | Atts >
   error(Result) .
---- non-empty error messages are forwarded
rl < X : Proxy | err: ERR, Atts >      ---- error
   received(X, ERR, Result)
=> < X : Proxy | Atts >
   if Result == "" then none else error(Result) fi . ---- no error message
---- sent and exited messages are consumed
```

```
  rl < X : Proxy | state: 7, io: IO, process: P, Atts >
     sent(X, IO)
     exited(X, P, normalExit(0))
  => none .
endm
```

The execution of the program is then as follows.

```
Maude> erew <> init .
result Configuration: <> result("104")
```

9.5 Control-C on external events

When a control-C is typed during regular rewriting, execution is suspended and Maude gets into its debugger. However, when Maude is suspended on an external event, because no internal rewrites are possible, and it receives a control-C, it prints a message rather than immediately aborting. A second control-C without intervening rewrites will cause an abort to the command line without any attempt to print the current state.

The reason for this is to leave open the possibility of recovering from potential problems happening on external objects. Following the conventions of the underlying operative systems, "Interrupt signals generated in the terminal are delivered to the active process group, which here includes both parent and child." In other words, when a control-C is typed, the signal is not only received by the Maude process, but also by all other active external processes depending on it. Each of these processes may respond to a control-C in different ways, but there might be cases in which we can make it recover once interrupted. If we do not want to try to recover the execution, and it remains suspended on the same event, we may type a second control-C to abort.

Let us illustrate some of these ideas in concrete examples. Specifically, let us revisit the examples presented in Section 9.4. To get the execution to suspend on an external event, let us introduce a little change in the specification of the desk calculator process example (module `PROCESS-DC` in Section 9.4.1). The file `process-dc-faulty.maude` includes exactly the same code as the one used in Section 9.4.1, but when submitting the second expression to be evaluated, instead of the message

```
send(IO, X, "4 * p\n")
```

it is being sent

```
send(IO, X, "4 * p")
```

Notice that without `\n`, the dc does not process the input, and therefore it does not respond to our subsequent `receive` message. The process gets suspended waiting for an answer that will never arrive. By typing a control-C while the process is suspended, we observe the following behavior:

```
Maude> erew <>
>     < me : User | state: 1, process: null, io: null, err: null, result: "" >
>     createProcess(processManager, me, "dc", nil, none) .
erewrite in PROCESS-DC :
   <>
   < me : User | state: 1, process: null, io: null, err: null, result: "" >
   createProcess(processManager, me, "dc", nil, none) .
^C
Control-C while suspended on external event(s).
```

```
A second control-C on the same suspension will abort execution and return to command
  line.
rewrites: 11 in 1ms cpu (4721ms real) (8560 rewrites/second)
result Configuration:
   <>
   < me : User |
     state: 6, process: process(58212), io: socket(3), err: socket(5), result: "" >
   signalProcess(process(58212), me, "SIGTERM")
   exited(me, process(58212), terminatedBySignal("SIGINT"))
Maude>
```

The control-C does not kill the Maude run. It shows the message

```
Control-C while suspended on external event(s).
A second control-C on the same suspension will abort execution and return to command
  line.
```

indicating what the situation was when the control-C was typed, and that a second control-C might be needed to abort. However, since an interrupt signal is sent to the dc process, it gets killed. This is the behavior of the dc application. As we will see below, other applications may respond differently to this kind of signals. But, since the external process terminates, the execution is no longer suspended. Indeed, as we can see in the final output provided, we are informed that the process exited due to the reception of a `SIGINT` signal.

Let us now consider our second example, the `PROCESS-MAUDE` specification in Section 9.4.2. We do not change anything on this example, just make it use the `PROCESS-DC` specification modified as above (available in file `process-maude-faulty.maude`). The question is that if we load the specification and run it with the corresponding `erew` command, we observe that the execution gets suspended. Notice that the program is creating a Maude process which loads the `process-dc-faulty.maude` file. The execution of the `erew` command gets suspended, as we saw above. The interesting thing is that we have just learnt how Maude handles an interrupt signal, and then, we know that if we type control-C, the external process also receives a control-C (an interrupt signal). Upon reception of this interrupt signal, the external Maude process produces some output, which is received by the main Maude process. It however does not receive a valid output, which leads to another suspension, although in this case it is the Python process that suspends, because it is unable to handle the text that receives. If a new control-C is typed, since it is a different suspension, the execution does not abort. Instead, we get a new message informing us about the situation, and telling us that, if the execution remains suspended in the same place, a second control-C will abort the execution. Since the interrupt signal received by the Python process does not get the execution to advance, a third control-C (second on this suspension event), forces the execution to abort. The following shows the interaction just explained:

```
erewrite in PROCESS-MAUDE : <> init .
^C
Control-C while suspended on external event(s).
A second control-C on the same suspension will abort execution and return to command line.
^C
Control-C while suspended on external event(s).
Note that this is a different suspension than the one that received a control-C 63 rewrites ago.
A second control-C on the same suspension will abort execution and return to command line.
^C
Second control-C while suspended on external event(s).
Aborting execution and returning to command line.
```

```
Advisory: closing open files.
Maude>
```

Note that no final output is presented. As we have seen above, if a control-C triggers some behavior in the external process that exits the suspension, Maude can complete its execution to a final state or end up suspended on some other external event which again requires two control-C events to abort. However, aborting execution is inherently a messy procedure that never has a printable final state.

Chapter 10

Strategy Language

Rule rewriting is a highly nondeterministic process: at every step, many rules could be applied at various positions. A finer control on rule application is sometimes desirable, and even required, when rules are not terminating or not convergent. Maude programmers already had resources to restrain rewriting, from adding more information to the data representation to applying rules explicitly at the metalevel, as described in Section 17.7. However, these methods make specifications harder to understand and programming at the metalevel is a cumbersome task for novice users. For these reasons, a strategy language has been proposed [97, 61, 99] as a specification layer above those of equations and rules. This provides a cleaner way to control the rewriting process respecting the *separation of concerns* principle. That is, the rewrite theory is not modified in any way: strategies provide an additional specification level above that of rules, so that the same system module may be executed according to different strategy specifications. The design of Maude's strategy language has been influenced, among others, by ELAN [13] and Stratego [18].

Of course, the most basic action of the strategy language is rule application, which is invoked by mentioning the rule label. More complex strategies, involving several, or unboundedly many, rule applications, can be built by means of various strategy combinators. The language is described in detail in Section 10.1. Moreover, strategy expressions can be given a name to be later invoked with arguments, even recursively. *Strategy modules*, the modular way in which strategies are declared, are explained in Section 10.2. Like functional and system modules, strategy modules can be parameterized, with specific features that are described in Section 10.3. A discussion about the strategy search commands follows in Section 10.4, and the chapter concludes with a case study on logic programming in Section 10.5. Some other aspects are covered in specific chapters, like the metarepresentation of strategies in Section 17.3 and how strategy executions can be traced and debugged in Section 20.

Let us start with a simple example. The `HANOI` module below specifies the Tower of Hanoi puzzle, invented by the French mathematician Éduard Lucas in 1883 [92]. His story tells about an Asian temple where there are three diamond posts. The first one is surrounded by sixty-four golden disks of increasing size from the top to the bottom. The monks are committed to move them from one post to another respecting two rules: only a disk can be moved at a time, and they can only be laid either on a bigger disk or on the floor. Their objective is to move all of them to the third post, and then the world will end.

```
mod HANOI is
  protecting NAT-LIST .

  sorts Post Hanoi Game .
```

```
  subsort Post < Hanoi .

  op (_) [_] : Nat NatList -> Post [ctor] .
  op empty  : -> Hanoi [ctor] .
  op __     : Hanoi Hanoi -> Hanoi [ctor assoc comm id: empty] .

  vars S T D1 D2 N : Nat .
  vars L1 L2     : NatList .
  vars H H'      : Hanoi .

  crl [move] : (S) [L1 D1] (T) [L2 D2] => (S) [L1] (T) [L2 D2 D1] if D2 > D1 .
  rl  [move] : (S) [L1 D1] (T) [nil] => (S) [L1] (T) [D1] .
endm
```

Here, the golden disks are modeled as natural numbers describing their size, and the posts are represented as lists of disks in bottom up order. We try to rewrite the initial puzzle setting

```
rew (0)[3 2 1] (1)[nil] (2)[nil] .
```

but the command does not terminate: the disks are being moved in a loop. To prevent such a situation, strategies can be useful. The Maude command for rewriting with strategies is:

`srewrite [`n`] in` ⟨*ModId*⟩ `:` ⟨*Term*⟩ `by` ⟨*StrategyExpression*⟩ `.`

It rewrites the term according to the given strategy expression and prints all the results. Since strategies need not be deterministic, many results may be obtained. Like in the standard rewriting commands, we can optionally specify the module where to rewrite after `in`, and a bound n on the number of solutions to be shown just after the command keyword, which can be shortened to `srew`. For example,

```
Maude> srew [3] in HANOI : (0)[3 2 1] (1)[nil] (2)[nil] using move .

Solution 1
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Hanoi: (0)[3 2] (1)[1] (2)[nil]

Solution 2
rewrites: 2 in 0ms cpu (1ms real) (~ rewrites/second)
result Hanoi: (0)[3 2] (1)[nil] (2)[1]

No more solutions.
rewrites: 2 in 0ms cpu (1ms real) (~ rewrites/second)
```

The two results of applying the `move` rule to the initial term are shown. Every time we request all solutions or more solutions than possible, the interpreter indicates that there are no more solutions when the search finishes. The order in which solutions appear is implementation-dependent. Following the output of the `srewrite` command, we can view a strategy expression α as a transformation $T_\Sigma \to \mathcal{P}(T_\Sigma)$ from an initial term t to a set of terms being the results of α in t. To elaborate more complex strategies, we need to introduce the complete strategy language.

10.1 The strategy language

As we have anticipated, rule application is the essential building block of the strategy language. Besides the rule label, further restrictions can be imposed. Its most general syntax has the form:

label[X_1 <- t_1, ..., X_n <- t_n]{α_1, ..., α_m}

Between square brackets, we can optionally set an initial ground substitution for the variables $X_1, \ldots, X_n$ appearing in the rule. And when invoking a conditional rule with rewriting conditions we must provide between curly brackets the strategies $\alpha_1, \ldots, \alpha_m$ to control rewriting each of them. Only rules with exactly m condition fragments will be tried, so all rewriting fragments must be given a strategy. For example, we can select to which Hanoi tower the disk is moved by giving the command:

```
Maude> srew (0)[3 2 1] (1)[nil] (2)[nil] using move[T <- 2] .

Solution 1
rewrites: 1
result Hanoi: (0)[3 2] (1)[nil] (2)[1]

No more solutions.
rewrites: 1
```

Constrained by the substitution, the rewriting engine tries to match with extension (see Section 4.8) against the lefthand sides `(S)[L1 D1] (2)[L2 D2]` for the first `move` rule, and `(S)[L1 D1] (2)[nil]` for the second `move` rule, but only the second succeeds and the term is rewritten to solution 1.

An additional combinator `all` triggers a single rewriting step with all rules available in the module, even those which are not given a label, but excluding those marked with `nonexec`, external objects and implicit rules. Rewriting condition fragments are executed like in the usual rewriting engine, without any restriction. To illustrate `all` and how strategies can control rule conditions, the following module `HANOI-COUNT` introduces a pair operator of `Game` for keeping track of the number of moves used to reach a solution.

```
mod HANOI-COUNT is
  protecting HANOI .

  op <_,_> : Hanoi Nat -> Game [ctor] .

  vars H H' : Hanoi .
  var  N : Nat .

 crl [step] : < H, N > => < H', s N > if H => H' .
  rl [cancel] : N => 0 [nonexec] .
  rl [inc] : N => s N [nonexec] .
endm
```

By controlling the `H => H'` condition of `step` with `all`, we indicate that only one rule application has to be performed in each game step, so that the count is calculated correctly. The other rules `cancel` and `inc` will be used later to count moves in a different way.

```
Maude> srew < (0)[3 2 1] (1)[nil] (2)[nil], 0 > using step{all} .

Solution 1
result Game: < (0)[3 2] (1)[1] (2)[nil],1 >

Solution 2
result Game: < (0)[3 2] (1)[nil] (2)[1],1 >

No more solutions
```

By default, rule applications are tried anywhere within the subject term. However, they can be limited to the topmost position using the `top` restriction. For example, if we apply the `cancel` rule, which rewrites any natural number[1] to zero, we obtain multiple solutions:

```
Maude> srew 1 using cancel .

Solution 1
result Zero: 0

Solution 2
result NzNat: 1

No more solutions.
```

In this case, the rule can be applied both on `1 = s 0`, producing `0`, and on its subterm `0`, producing `s 0`. Instead, using `top` the rule is only applied at the top position `s 0`:

```
Maude> srew 1 using top(cancel) .

Solution 1
result Zero: 0

No more solutions.
```

Combining `top` with the `matchrew` operator described in Section 10.1.2, rules can be applied to any specified subterm in the subject term.

The other basic component of the language are the tests. They can be used for testing a condition on the subject term. Their syntax has the form:

`match` P `s.t.` C

where P is a pattern and C is an equational condition, as described in Section 4.3. Their syntax and behavior are similar to those of the `match` commands in Section 4.9: the such-that clause can be omitted if no condition is imposed, and the starting keyword indicates where the matching is done: on top (`match`), on any fragment of the flattened top modulo axioms (`xmatch`), or anywhere within the subject term (`amatch`). The condition may refer to variables in the pattern, whose value will be obtained from the matching. On a successful match and condition check, the result is the initial term. Otherwise, the test does not provide any solution. For example, we can check whether the towers of Hanoi puzzle is solved with tests:

```
Maude> srew (0)[nil] (1)[nil] (2)[3 2 1] using match (N)[3 2 1] H s.t. N =/= 0 .

Solution 1
result Hanoi: (0)[nil] (1)[nil] (2)[3 2 1]

No more solutions.

Maude> srew (0)[nil] (1)[nil] (2)[3 2 1] using xmatch (0)[nil] (2)[3 2 1] .

Solution 1
result Hanoi: (0)[nil] (1)[nil] (2)[3 2 1]

No more solutions.
```

[1] The decimal representation of natural numbers is just syntactic sugar for their Peano notation, whose constructors are `0` and the successor `s`, as explained in Section 7.2. Hence, a natural number is a tower of `s` symbols followed by a `0`, which are efficiently supported using the `iter` attribute (see Section 4.4.2).

Strategy ζ	Results $[\![\zeta]\!](\theta, t)$
`idle`	$\{t\}$
`fail`	$\emptyset$
$rlabel[\rho]$	$\{\ t' \in T_\Sigma \mid t \to_{\rho(l)\to\rho(r)} t' \text{ for any } l \to^{rlabel} r \in R\ \}$
$\alpha\,;\beta$	$\bigcup_{t' \in [\![\alpha]\!](\theta,t)} [\![\beta]\!](\theta, t')$
$\alpha \,\mathtt{|}\, \beta$	$[\![\alpha]\!](\theta, t) \cup [\![\beta]\!](\theta, t)$
α`*`	$\bigcup_{n=0}^{\infty} [\![\alpha]\!]^n(\theta, t)$
`match` P `s.t.` C	$\begin{cases} \{t\} & \text{if matches}(P, t, C, \theta) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$
α `?` β `:` γ	$\begin{cases} [\![\alpha\,;\beta]\!](\theta, t) & \text{if } [\![\alpha]\!](\theta, t) \neq \emptyset \\ [\![\gamma]\!](\theta, t) & \text{if } [\![\alpha]\!](\theta, t) = \emptyset \end{cases}$
`matchrew` P `s.t.` C `by` X_1 `using` α_1`, ...,` X_n `using` α_n	$\bigcup_{\sigma \in \text{matches}(P,t,C,\theta)} \Big(\bigcup_{t_1 \in [\![\alpha_1]\!](\sigma,\sigma(X_1))} \cdots \bigcup_{t_n \in [\![\alpha_n]\!](\sigma,\sigma(X_n))} \sigma[x_1/t_1, \ldots, x_n/t_n](P) \Big)$
$slabel(t_1, \ldots, t_n)$	$\bigcup_{(lhs,\delta,C) \in \text{Defs}} \bigcup_{\sigma \in \text{matches}(slabel(t_1,\ldots,t_n), lhs, C, \text{id})} [\![\delta]\!](\sigma, t)$

Table 10.1: Main strategy combinators and their informal semantics

Since the `xmatch` tests matches inside the top symbol with extension, and the two posts in the pattern form a fragment of the associative and commutative symbol on the subject term, the second command succeeds. The same command `match` will fail the match, because the pattern is not the whole term.

```
Maude> srew (0)[nil] (1)[nil] (2)[3 2 1] using match (0)[nil] (2)[3 2 1] .

No solution.
```

With the `amatch` variant, we can check whether the disks of radius 3 and 1 are the same post with the following command:

```
Maude> srew in HANOI : (0)[nil] (1)[nil] (2)[3 2 1] using amatch 3 L1 1 .

Solution 1
result Hanoi: (0)[nil] (1)[nil] (2)[3 2 1]

No more solutions.
```

Once we have rule application and tests, the basic actions and control of the language, we can combine them to build more complex strategies. Table 10.1 summarizes all primitive strategy constructors and informally describes their semantics[2].

10.1.1 Basic control combinators

To control the flow of rule execution some combinators are defined. Let α, β and γ represent arbitrary strategy expressions.

[2]We denote by $[\![\zeta]\!](\theta, t)$ the results of evaluating the strategy expression ζ in the term t with a variable context substitution θ, by matches(P, t, C, θ) the matches of the term t into the pattern P satisfying the condition C and extending the bindings of the previous substitution θ, and by Defs the set of strategy definitions. In the remainder of this section, we will explain in detail all these combinators.

1. The concatenation $\alpha;\beta$ executes the strategy α and then the strategy β on each α result. In our example, we can move twice using the strategy `move ; move`.

```
Maude> srew (0)[3 2 1] (1)[nil] (2)[nil] using move ; move .

Solution 1
result Hanoi: (0)[3 2 1] (1)[nil] (2)[nil]

Solution 2
result Hanoi: (0)[3] (1)[1] (2)[2]

Solution 3
result Hanoi: (0)[3 2] (1)[nil] (2)[1]

Solution 4
result Hanoi: (0)[3] (1)[2] (2)[1]

Solution 5
result Hanoi: (0)[3 2] (1)[1] (2)[nil]

No more solutions.
```

   Notice that the initial state appears first in the solutions. This suggest why the initial `rewrite` command did not terminate.

2. The disjunction or alternative $\alpha\,|\,\beta$ executes α or β. In other words, the results of $\alpha\,|\,\beta$ are both those of α and those of β.

```
Maude> srew (0)[3 2] (1)[1] (2)[nil] using move[S <- 0] | move[T <- 0] .

Solution 1
result Hanoi: (0)[3 2 1] (1)[nil] (2)[nil]

Solution 2
result Hanoi: (0)[3] (1)[1] (2)[2]

No more solutions.
```

3. The iteration α`*` runs α zero or more times consecutively. If we issue the command

```
Maude> srew (0)[3 2 1] (1)[nil] (2)[nil] using move * .

Solution 1
result Hanoi: (0)[3 2 1] (1)[nil] (2)[nil]
...

Solution 27
result Hanoi: (0)[nil] (1)[1] (2)[3 2]

No more solutions.
```

   we obtain the 27 admissible states, where the disks in each post are in the correct order.

4. The concatenation, alternative, and iteration combinators resemble the similar constructors for regular expressions. The empty word and empty language constants are here

represented by the `idle` and `fail` operators. The result of applying `idle` is always the initial term, while `fail` generates no solution.

```
Maude> srew (0)[3 2 1] using idle .

Solution 1
result Post: (0)[3 2 1]

No more solutions.

Maude> srew (0)[3 2 1] using fail .

No solution.
```

In a wider sense, we say that a strategy *fails* when no solution is obtained. Failures can happen in less explicit situations like vacuous rule applications, unsatisfied tests, etc.

5. A conditional strategy is written α `?` β `:` γ. It executes α and then β on its results, but if α does not produce any, it executes γ on the initial term. That is, α is the *condition*; β the *positive branch*, which applies to the results of α; and γ the *negative branch*, which is applied only if α fails.

   For example, we can instruct the monks to move disks to the first and second tower only if no disk can be moved to the target post. In this way, we obtain only 21 results out of 27.

```
Maude> srew in HANOI : (0)[3 2] (1)[1] (2)[nil] using
        (move[T <- 2] ? idle : move) * .

Solution 1
result Hanoi: (0)[3 2] (1)[1] (2)[nil]
...

Solution 21
result Hanoi: (0)[nil] (1)[1] (2)[3 2]

No more solutions.
```

In the example above, we are giving precedence to one strategy over another, so that the second is only executed if the first fails. This is a common control pattern, and so it is defined as a derived operator with its own name, along with some other usual constructions:

- The `or-else` combinator is defined by

  $$\alpha \texttt{ or-else } \beta \equiv \alpha \texttt{ ? idle : } \beta$$

  As shown in the example above, it executes β only if α has failed.

- The *non-void iteration* α`+` is defined as $\alpha\texttt{+} \equiv \alpha \texttt{ ; } \alpha\texttt{*}$.

- The *negation* is defined as:

  $$\texttt{not(}\alpha\texttt{)} \equiv \alpha \texttt{ ? fail : idle}$$

  It fails when α succeeds, and succeeds as an `idle` when α fails.

- The *normalization operator*

$$\alpha\texttt{!} \equiv \alpha\texttt{* ; not(}\alpha\texttt{)}$$

applies α until it cannot be further applied.

```
Maude> srew (0)[3 2 1] (1)[nil] (2)[nil] using move[S <- 0] ! .

Solution 1
result Hanoi: (0)[3] (1)[1] (2)[2]

Solution 2
result Hanoi: (0)[3] (1)[2] (2)[1]

No more solutions.
```

- The `try` strategy is defined by `try(`α`)` $\equiv \alpha$ `? idle : idle`. It applies α, but if it fails, it returns the initial state.
- The `test` strategy, defined by `test(`α`)` $\equiv$ `not(not(`α`))`, fails whenever α fails, but when α succeeds, it returns the initial term instead of its results.

```
Maude> srew (0)[3] (1)[1] (2)[2] using test(move[S <- 0]) .

No solution.

Maude> srew (0)[3] (1)[1] (2)[2] using test(move[S <- 1]) .

Solution 1
result Hanoi: (0)[3] (1)[1] (2)[2]

No more solutions.
```

10.1.2 Rewriting of subterms

The *match and rewrite* operator `matchrew` restricts the application of a strategy to a specific subterm of the subject term. Its syntax is

`matchrew` $P(X_1,\ldots,X_n)$ `s.t.` C `by` X_1 `using` α_1`, ...,` X_n `using` α_n

where P is a pattern with variables $X_1,\ldots,X_n$ among others, and C is an optional equational condition. The `using` clauses associate variables in the pattern, which are matched by subterms of the matched term, with strategies that will be used to rewrite them. These variables must be distinct and must appear in the pattern. As in the case of tests, there are three flavors for this operator, depending on the type of matching: `matchrew`, `xmatchrew`, and `amatchrew`.

The semantics of this operator is illustrated in Figure 10.1. All matches of the pattern in the subject term are tried, checking the condition if any. If none succeeds, the strategy fails. Otherwise, for each match of the main pattern the subterms matching each X_i are extracted, rewritten according to α_i, and their solutions are put in place of the original subterm. The rewriting of the subterms happens in parallel, in the sense that the terms evolve in a completely independent and fair way. Finally, all possible combinations of the subterm results will give rise to different reassembled solutions.

Another advantage of `matchrew` is that it let us obtain information about the term and use it to modify the behavior of the strategies. The scope of the variables in the pattern and the condition is extended to the substrategies $\alpha_1,\ldots,\alpha_n$, so that the values they take by

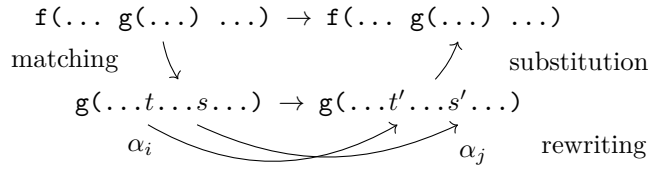


Figure 10.1: Behavior of the `amatchrew` combinator

matching and evaluation can be used in nested rule substitutions, strategy calls, and `match` and `matchrew` operators. Conversely, when these variables are bound by outer constructs, they keep their values and matching is done against a partially instantiated pattern. When a variable in the condition is not bound either by the outer context or by the matching, the strategy will be unusable and a warning will be shown by the interpreter.

For example, instead of using the `step` rule to count the moves in the Tower of Hanoi, we could have used a `matchrew` and the increment rule `inc`. This is written:

```
Maude> srew [1] in HANOI-COUNT : < (0)[3 2 1] (1) [nil] (2)[nil], 0 >
        using (matchrew < H, N > by H using move, N using top(inc)) * ;
          amatch (2)[3 2 1] .

Solution 1
rewrites: 523 in 56ms cpu (56ms real) (9339 rewrites/second)
result Game: < (0)[nil] (1)[nil] (2)[3 2 1], 7 >
```

Notice the use of `top` in the `inc` rule application. As we have seen in the previous example with the `cancel` rule in page 262, `inc` could have been applied inside successors if we had omitted the `top` modifier. Although in this particular case the result would not have been affected, unnecessary work would have been done, as we can observe in the rewrites count of the following execution.

```
Maude> srew 1 using inc .

Solution 1
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 2

No more solutions.
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
```

10.1.3 The `one` operator

Strategies are nondeterministic due to the different possible matches of the rules and the subterm rewriting operator, disjunction, iteration, etc. The various rewriting paths that these alternatives make possible are explored almost simultaneously by the strategic search engine. Sometimes, the different options lead to the same result or, the results being different, it may not matter which of them is selected. For efficiency purposes, the `one` combinator is introduced to execute its argument strategy and return only one of its solutions, namely the first one found. If the strategy does not produce any solutions, neither does `one`.

```
Maude> srew (0)[3 2 1] (1)[nil] (2)[nil] using one(move * ; amatch (2)[3 2 1]) .
```

```
Solution 1
rewrites: 121
result Hanoi: (0)[nil] (1)[nil] (2)[3 2 1]

No more solutions.
rewrites: 121

Maude> srew (0)[3 2 1] (1)[nil] (2)[nil] using move * ; amatch (2)[3 2 1] .

Solution 1
rewrites: 121
result Hanoi: (0)[nil] (1)[nil] (2)[3 2 1]

No more solutions.
rewrites: 150
```

The most noticeable performance improvements are obtained the closer `one` is applied to rules, because the alternatives are pruned immediately. However, this should be done only when paths can be discarded safely.

```
Maude> srew (0)[3 2 1] (1)[nil] (2)[nil] using one(move) * .

Solution 1
result Hanoi: (0)[3 2 1] (1)[nil] (2)[nil]

Solution 2
result Hanoi: (0)[3 2] (1)[1] (2)[nil]

No more solutions.
```

In the execution above, `one(move)` tries only a single move at each iteration, and the resulting sequence of moves does not lead to the solved game (in fact, it loops after the first move).

10.1.4 Strategy calls

Strategies with their own name and taking parameters can be defined in *strategy modules*, which will be explained in Section 10.2. These named strategies can be called as a strategy expression with a typical prefix operator syntax: a strategy identifier followed by a comma-separated list of arguments between parentheses. When the strategy does not take any arguments and there is no rule with the same name, the parentheses can be omitted. Suppose we already have defined a strategy `moveAll` to move disks between certain posts indicated by its arguments, with its third argument being the number of disks to be moved. It can be applied to an initial state like this:

```
Maude> srew (0)[3 2 1] (1)[nil] (2)[nil] using moveAll(0, 2, 3) .

Solution 1
rewrites: 24 in 4ms cpu (2ms real) (6000 rewrites/second)
result Hanoi: (0)[nil] (1)[nil] (2)[3 2 1]

No more solutions.
rewrites: 24 in 4ms cpu (2ms real) (6000 rewrites/second)
```

How this and other strategies are defined is explained in the next section.

10.2 Strategy modules

Strategy modules let the user give name to strategy expressions. Named strategies facilitate the use and description of complex strategies, and they extend the expressiveness of the language by means of recursive and mutually recursive strategies. As for functional and system modules, a strategy module is declared in Maude using the keywords

`smod` ⟨*ModuleName*⟩ `is` ⟨*DeclarationsAndStatements*⟩ `endsm`

A typical strategy module would import the system module it will control, declare some strategies, and define them by means of strategy definitions. All statements and declarations that were available in functional and system modules can also be included in strategy modules, but we encourage to avoid them as much as possible to emphasize a clean separation between the rewriting theory and the strategies that control it.

For example, the following strategy module `HANOI-SOLVE` defines the recursive strategy `moveAll` that we have used in Section 10.1.4 to illustrate strategy calls and solve the puzzle.

```
smod HANOI-SOLVE is
  protecting HANOI .
  protecting HANOI-AUX .

  strat moveAll : Nat Nat Nat @ Hanoi .

  vars S T C M : Nat .

  sd moveAll(S, S, C) := idle .
  sd moveAll(S, T, 0) := idle .
  sd moveAll(S, T, s(C)) := moveAll(S, third(S, T), C) ;
                            move[S <- S, T <- T] ;
                            moveAll(third(S, T), T, C) .
endsm
```

The operator `third`, imported from the following functional module `HANOI-AUX`, calculates the third with respect to two given posts. In general, when we require some auxiliary operations to specify a strategy, we can define them in an auxiliary module that is then imported by the strategy module.

```
fmod HANOI-AUX is
  protecting SET{Nat} .

  op third : Nat Nat ~> Nat .

  vars N M K : Nat .

  ceq third(N, M) = K if N, M, K := 0, 1, 2 .
endfm
```

The features making up a strategy module are strategy declarations and definitions. Strategies are declared as follows:

`strat` ⟨*StratName*⟩+ `:` ⟨*Sort-1*⟩ ... ⟨*Sort-k*⟩ `@` ⟨*Sort*⟩ `[`⟨*StratAttributes*⟩`]` .

Strategy names are single word identifiers. The sorts between the colon and the `@` sign are the argument sorts of the strategy. The last argument after the `@` sign is the *subject sort*, i.e., the sort of the terms to which the strategy will be applied. The colon can be omitted when a strategy has no arguments. Multiple strategy names, separated by blank space, can be used to

define multiple strategies at once, provided that all have the same argument sorts and subject sort. In this case the plural keyword `strats` instead of `strat` is recommended. The only admitted strategy attribute is `metadata` (see Section 4.5.2).

Strategy names can be *overloaded*, i.e., strategies can be defined with the same name but may have different arities and argument types. Like operators, strategies are defined at the kind level. Homonymous strategies with the same arity and such that for each argument the corresponding sorts have the same kind are considered the same. In practice, the concrete sorts of the strategy signature do not have any effect and should be considered as a declaration of intent. Repeated declarations of the exact same strategy twice (with the same name and argument kinds) will generate a warning. For example, when processing the module

```
smod STRATS is
  protecting INT .

  strat repeat : Nat @ Nat .
  strat repeat : Int @ Nat .
endsm
```

the interpreter will issue the warning:

```
Warning: <standard input>, line 5 (smod STRATS): strategy declaration
  strat repeat : Int @ Nat . conflicts with
  strat repeat : Nat @ Nat . from <standard input>, line 4 (smod STRATS).
```

For its part, the subject sort is ignored to all effects: it is not checked when calling the strategy and not used to distinguish strategies.

Without a corresponding definition, named strategies behave like `fail`. Definition statements should be provided to properly define each named strategy according to the syntax:

`sd` ⟨*StrategyCall*⟩ `:=` ⟨*StrategyExpression*⟩ `[`⟨*SdAttributes*⟩`]` `.`

The lefthand side of a definition is a strategy call, as seen in the previous section, probably containing variables, which can be used in the righthand side. In fact, all variables in the strategy expression, except variables in matching patterns, must appear in the lefthand side. Like equations and rules, strategy definitions may be conditional:

`csd` ⟨*StrategyCall*⟩ `:=` ⟨*StrategyExpression*⟩ `if` ⟨*Condition*⟩ `[`⟨*SdAttributes*⟩`]` `.`

The condition of a strategy definition is an equational one, as described in Section 4.3, i.e., rewriting conditions t_1 `=>` t_2 are not allowed. The attributes `metadata`, `label` and `print` (see Section 4.5) can be attached to `sd` and `csd` definitions. When a strategy is invoked, all available definitions for the name are tried, and all those whose lefthand side matches the calling arguments are executed. Moreover, conditions may generate different variable assignments, in which case the strategy will be applied with each one of them. For example, we could have given the third definition of `moveAll` in `HANOI-SOLVE` above as

```
csd moveAll(S, T, s(C)) := moveAll(S, third(S, T), C) ;
                           move[S <- S, T <- T] ;
                           moveAll(third(S, T), T, C) if S =/= T .
```

since the definition does not make sense when the target and source towers are the same. In fact, this case always fails because the partial function `third(S, T)` evaluates to an error term of sort `[Nat]` such that the first recursive call does not match any definition.

To conclude this section, let us show another example: the *insertion-sort algorithm* implemented by means of strategies. The underlying array is represented as a set of pairs containing

both a position and a value. It is assumed that there is a single entry for each position between one and the length of the array. A rule `swap` is defined to swap the values between two entries.

```
mod SWAPPING{X :: DEFAULT} is
  protecting ARRAY{Nat, X} .
  vars I J : Nat .
  vars V W : X$Elt .
  var AR : Array{Nat, X} .
  rl [swap] : J |-> V ; I |-> W => J |-> W ; I |-> V .

  op maxIndex : Array{Nat, X} ~> Nat .
  eq maxIndex(empty) = 0 .
  eq maxIndex(I |-> V ; AR) = if maxIndex(AR) < I then I else maxIndex(AR) fi .
endm
```

Then, the insertion-sort algorithm is specified as the `insort` strategy. It visits each position of the array in increasing order, maintaining the invariant that the entries to the left of the current index are sorted. At each iteration, it inserts the current value in the sorted fragment of the array. Insertion is done by the `insert` strategy, which compares in descending order each entry with its neighbor and swaps them if they are misplaced. The base case is the insertion in the first position, which does not require any work.

```
view DEFAULT+ from DEFAULT to STRICT-TOTAL-ORDER + DEFAULT is
endv


smod INSERTION-SORT{X :: STRICT-TOTAL-ORDER + DEFAULT} is
  protecting SWAPPING{DEFAULT+}{X} * (
    sort Array{Nat, DEFAULT+}{X} to NatArray{X}
  ) .

  strat swap : Nat Nat @ NatArray{X} .
  strats insert insort : Nat @ NatArray{X} .

  vars X Y J I : Nat .
  vars V W : X$Elt .
  var AR : NatArray{X} .

  sd insort(Y) := try(match AR s.t. Y <= maxIndex(AR) ;
                      insert(Y) ;
                      insort(Y + 1)) .

  sd insert(1) := idle [label base-case] .
 csd insert(s(X)) := try(xmatch X |-> V ; s(X) |-> W s.t. W < V ;
                         swap(X, s(X)) ;
                         insert(X))
                      if X > 0 [label recursive-case] .

  sd swap(X, Y) := swap[J <- X, I <- Y] .
endsm


view Int<0 from STRICT-TOTAL-ORDER + DEFAULT to INT is
  sort Elt to Int .
endv
```

```
smod INSERTION-SORT-INT is
  protecting INSERTION-SORT{Int<0} .
endsm
```

Then, we can sort any array using `insort` with the following `srewrite` command:

```
Maude> srewrite 1 |-> 8 ; 2 |-> 3 ; 3 |-> 15 ; 4 |-> 5 ; 5 |-> 2 using insort(2) .

Solution 1
rewrites: 116 in 0ms cpu (0ms real) (~ rewrites/second)
result NatArray{Int<0}: 1 |-> 2 ; 2 |-> 3 ; 3 |-> 5 ; 4 |-> 8 ; 5 |-> 15

No more solutions.
rewrites: 116 in 0ms cpu (1ms real) (~ rewrites/second)
```

10.2.1 Module importation

Strategy modules can be seen as an additional layer of specification over functional and system modules, which are not allowed to import strategy modules. Instead, strategy modules can import functional, system, or strategy modules using the `protecting`, `extending`, and `including` modes.

The semantic requirements for each inclusion mode are described in Section 6.1 for functional and system modules. Similar requirements apply to strategies when importing strategy modules. A `protecting` extension M of a strategy module M' should not alter the rewriting paths permitted by each strategy declared in M'. The declaration and definition of new strategies in the importing module does not affect the semantics of the strategies declared in the imported module, but adding new definitions for the imported strategies may change their behavior. If new rewriting paths are added to the imported strategies, but those of the imported module are preserved, the `extending` mode should be used. In any other case, the `including` mode is the right choice.

Let us illustrate the importation modes with a simple example. The following strategy module `SMOD-IMPORT-EXAMPLE` declares two strategies `st1` and `st2`. Since `st1` is not given any definition, no rewriting is possible using it, and thus `st1` is equivalent to `fail`. On the other hand, the definition of `st2` describes all rewriting paths of the form $q \to$ `'z` for every quoted identifier q.

```
mod SMOD-IMPORT-EXAMPLE0 is
  protecting QID .
  rl [ab] : 'a => 'b .
  rl [bc] : 'b => 'c .
  rl [zz] : Q:Qid => 'z .
endm

smod SMOD-IMPORT-EXAMPLE is
  protecting SMOD-IMPORT-EXAMPLE0 .
  strats st1 st2 @ Qid .
  sd st2 := st1 ? bc : zz .
endsm
```

The module `SMOD-IMPORT-EXAMPLE'` below includes a new definition for `st2` that allows the additional rewriting path `'a` $\to$ `'b`. Hence, `SMOD-IMPORT-EXAMPLE` should not be imported in `protecting` mode but in `extending` mode.

```
smod SMOD-IMPORT-EXAMPLE' is
  extending SMOD-IMPORT-EXAMPLE .
  sd st2 := ab .
endsm
```

Although there is no direct way of restricting an already defined strategy, this may indirectly happen due to the conditional and its derived operators. The following module extends `st1` with the `'a` → `'b` path. As a result, the condition of `st1`'s definition `st1 ? bc : zz` has now solutions for `'a`, the rewriting path `'a` → `'b` → `'c` is allowed, but also `'a` → `'z` is lost, because the negative branch of the conditional is not evaluated.

```
smod SMOD-IMPORT-EXAMPLE'' is
  including SMOD-IMPORT-EXAMPLE .
  sd st1 := ab .
endsm
```

The semantic requirements for strategies refer to the rewriting paths that can be followed while executing the strategy, which is stronger than refering only to their evaluation results. However, it this example, the results are also changed non-conservatively:

```
Maude> srew in SMOD-IMPORT-EXAMPLE : 'a using st2 .

Solution 1
rewrites: 1
result Qid: 'z

No more solutions.
rewrites: 1

Maude> srew in SMOD-IMPORT-EXAMPLE'' : 'a using st2 .

Solution 1
rewrites: 2
result Qid: 'c

No more solutions.
rewrites: 2
```

As for functional and system modules, these requirements are not checked by Maude at runtime.

Finally, the language of module renamings (see Section 6.2.2) is extended with strategy renamings. Like for regular operators, `strat` *name* `to` *newName* renames all strategies whose name is *name* to *newName*. To rename only a specific instance of an overloaded strategy name, its arity and subject sort can be made explicit with `strat` $name : s_1 \dots s_n$ `@` s `to` *newName*.

10.3 Parameterization in strategy modules

Strategy modules can be parameterized like any other module in Maude, by means of theories and views, as explained in Section 6.3. A *parameterized strategy module* is a usual strategy module that takes a set of formal parameters, bound to some theories:

```
smod SM{X1 :: T1, ..., XN :: TN} is ... endsm
```

These theories can be functional or system theories, or also *strategy theories*. A strategy theory specifies the signature of the strategies that a concrete strategy module parameter must provide and the requirements that these actual strategies must satisfy. How a particular strategy module

meets the requirements of a strategy theory is specified by means of a view with strategy bindings, which will be described soon, and then views are used to instantiate parameterized modules with actual parameters. Strategy theories are introduced with the syntax:

```
sth T is ... endsth
```

and have features entirely similar to those of strategy modules. They can:

- import other theories in `including` mode,
- import modules of any kind,
- declare strategies,
- provide strategy definitions, and
- include other declarations and statements that functional and system theories admit.

The simplest strategy theory `STRIV` declares a single strategy without parameters:

```
sth STRIV is
   including TRIV .
   strat st @ Elt .
endsth
```

The subject sort `Elt` of the strategy `st` is imported from the functional theory `TRIV`. Strategy theories can never be used to parameterize functional or system modules.

Given a strategy module that meets the requirements of a strategy theory, we should specify how they are actually met by using a view. The repertoire of mappings that can appear in a view (see Section 6.3.2) has been extended with additional syntax for strategy bindings:

1. Mapping all formal strategies with the same given name at once. Each affected strategy will be seeked in the target module after translating its arguments according to the sort mappings of the view.

   `strat` ⟨*StratName*⟩ `to` ⟨*StratName*⟩ `.`

   For example, consider the following strategy theory `FOO` and strategy module `BAR`:

   ```
   sth STHEORY is
     including TRIV .
     protecting NAT .
     strat foo : Elt @ Elt .
     strat foo : Elt Nat @ Elt .
   endsth

   smod SMODULE is
     protecting STRING .
     protecting NAT .
     strat bar : String @ String .
     strat bar : String Nat @ String .
   endsm
   ```

   The following view `Bar` maps both the strategy `foo(Elt)` to `bar(String)` and the strategy `foo(Elt, Nat)` to `bar(String, Nat)` with a single mapping `strat foo to bar`.

```
view SModule from STHEORY to SMODULE is
  sort Elt to String .
  strat foo to bar .
endv
```

2. Mapping a single formal strategy to an existing strategy in the target module.

   `strat` ⟨*StratName*⟩ `:` ⟨*Sort-1*⟩ `...` ⟨*Sort-k*⟩ `@` ⟨*Sort*⟩ `to` ⟨*StratName*⟩ `.`

   For example, the previous view `Bar` can also be written as:

```
view SModule' from STHEORY to SMODULE is
  sort Elt to String .
  strat foo : Elt @ Elt to bar .
  strat foo : Elt Nat @ Elt to bar .
endv
```

3. Mapping a single formal strategy to an actual strategy expression in the target module.

   `strat` ⟨*StrategyCall*⟩ `to expr` ⟨*StrategyExpression*⟩ `.`

   Only variables are allowed as arguments in the lefthand side of the mapping, as for the operator `to term` mappings (see Section 6.3.2). These variables can be used in the strategy expression in the righthand side. When the module is instantiated, the strategy calls matching the lefthand side will be substituted by the lefthand side expression in any strategy definition of the module.

Finally, and as usual, modules are instantiated with views using the syntax

`SM{`$\texttt{View}_1$`, ..., `$\texttt{View}_N$`}`

Let us illustrate how parameterized strategy modules are instantiated by strategy views with an example. The following parameterized strategy module `BACKTRACKING` is a backtracking problem solver: the generic algorithm is specified as a strategy `solve` depending on some formal declarations.

```
smod BACKTRACKING{X :: BT-STRAT} is
  strat solve @ X$State .
  var S : X$State .
  sd solve := (match S s.t. isSolution(S)) ?
                idle : (expand ; solve) .
endsm
```

This module is parameterized by the backtracking problem specification, whose requirements are expressed in the strategy theory `BT-STRAT`. Its main element is the `expand` strategy, which is intended to generate all the successors of the problem state in which it is applied.

```
fth BT-ELEMS is
  protecting BOOL .
  sort State .
  op isSolution : State -> Bool .
endfth

sth BT-STRAT is
  including BT-ELEMS .
  strat expand @ State .
endsth
```

The strategy theory includes a functional theory `BT-ELEMS`, where we require a sort `State` for the problem states, and a predicate `isSolution` to recognize solutions among them. The strategy theory itself declares the strategy `expand` without parameters. Now, we specify a concrete backtracking problem, the 8-queens problem, and instantiate the parameterized `BACKTRACKING` module with it by means of a view. Then, the instantiated strategy `solve` will be able to solve the 8-queens problem. The problem is described in the system module `QUEENS` and the strategy module `QUEENS-STRAT`:

```
mod QUEENS is
  protecting LIST{Nat} .
  protecting SET{Nat} .

  op isOk : List{Nat} -> Bool .
  op ok : List{Nat} Nat Nat -> Bool .
  op isSolution : List{Nat} -> Bool .

  vars N M Diff : Nat .
  var  L : List{Nat} .
  var  S : Set{Nat} .

  eq isOk(L N) = ok(L, N, 1) .
  eq ok(nil, M, Diff) = true .
  ceq ok(L N, M, Diff) = ok(L, M, Diff + 1)
   if N =/= M /\ N =/= M + Diff /\ M =/= N + Diff .
  eq isSolution(L) = size(L) == 8 .

  crl [next] : L => L N if N, S := 1, 2, 3, 4, 5, 6, 7, 8 .
endm

smod QUEENS-STRAT is
  protecting QUEENS .

  strat expand @ List{Nat} .
  var L : List{Nat} .
  sd expand := top(next) ; match L s.t. isOk(L) .
endsm
```

Then, we define a view from the `BT-STRAT` strategy theory to `QUEENS-STRAT`:

```
view QueensBT from BT-STRAT to QUEENS-STRAT is
  sort State to List{Nat} .
  strat expand to expand .
endv
```

As with sorts and operators, unmentioned strategy names get the identity mapping, so the mapping for `expand` in `QueenBT` could be omitted. Finally, we can instantiate the parameterized module:

```
smod BT-QUEENS is
  protecting BACKTRACKING{QueensBT} .
endsm
```

and run `solve` for 8-queens:

```
Maude> srew [2] nil using solve .
srewrite [2] in BT-QUEENS : nil using solve .
```

```
Solution 1
rewrites: 285984 in 238ms cpu (238ms real) (1200211 rewrites/second)
result NeList{Nat}: 1 5 8 6 3 7 2 4

Solution 2
rewrites: 285984 in 238ms cpu (238ms real) (1200211 rewrites/second)
result NeList{Nat}: 1 6 8 3 7 4 2 5
```

Typically, views will be defined from a strategy theory to a strategy module or theory, but other combinations are possible. In general, strategic and non-strategic theories and modules can be combined with freedom in views.

1. Views from strategy theories to non-strategy modules (or theories) are useful when the target strategies are simple enough to be defined by strategy expressions, for which `to expr` bindings must be used.

2. Views from non-strategy theories to strategy modules (or theories) can be used to instantiate modules of a lower level with strategy modules. In that case, the instantiated module is automatically promoted to a strategy module. These views cannot contain strategy bindings, as there are no formal strategies to bind.

Otherwise, all the details about module instantiation in Section 6.3.4 are applicable to the strategy level as well. As a simple example, we can write a view from the functional theory `TRIV` to the strategy theory `STRIV` and vice-versa as follows:

```
view StrivIdle from STRIV to TRIV is
  sort Elt to Elt .
  strat st to expr idle .
endv

view STRIV from TRIV to STRIV is
  sort Elt to Elt .
endv
```

Moreover, we could have defined a view from the strategy theory `BT-STRAT` directly to the functional module `QUEENS` by binding the formal strategy `expand` to the actual inline definition of the `expand` strategy we already use in the strategy module `QUEENS-STRAT`. In this way, we do not need to define or use the `QUEENS-STRAT` strategy module.

```
view QueensBT2 from BT-STRAT to QUEENS is
  sort State to List{Nat} .
  var L : State .
  strat expand to expr top(next) ; match L s.t. isOk(L) .
endv
```

10.4 Strategy search and the `dsrewrite` command

Although strategies control and restrict rewriting, they do not make the process deterministic. A strategy may allow multiple rewriting paths, produced by alternative local decisions that may appear during its execution. The `rewrite` and `frewrite` commands eliminate this nondeterminism by each choosing a specific fixed strategy, as explained in Sections 5.4.1 and 5.4.2. Instead, the `search` command explores all the possible rule applications looking for a term

matching a given target. The `srewrite` command agrees with `search` in the exhaustive exploration of the alternatives, looking, in this case, for strategy solutions. However, the `srewrite` search is not conducted on the complete rewriting tree of `search`, but in a subtree pruned by the effect of the strategy. How this tree is explored has implications for the command's output and its performance.

The `srewrite` command explores the rewriting graph following a fair policy which ensures that all solutions reachable in a finite number of steps are eventually found, unless the interpreter runs out of memory. Without being a breadth-first search, multiple alternative paths are explored in parallel. This can be observed in the n-queens problem presented in Section 10.3, and in the particular example we showed:

```
Maude> srew [2] nil using solve .

Solution 1
rewrites: 285984 in 249ms cpu (248ms real) (1147935 rewrites/second)
result NeList{Nat}: 1 5 8 6 3 7 2 4

Solution 2
rewrites: 285984 in 249ms cpu (249ms real) (1147935 rewrites/second)
result NeList{Nat}: 1 6 8 3 7 4 2 5
```

Notice that both solutions are obtained at the same time and with the same number of rewrites, since these and even more rewriting paths are being explored in parallel. All of them increment indiscriminately the common rewrites counter. In this particular example, being all the solutions at the same depth in the rewriting tree, the tree has to be explored almost completely before the first solution is found.

An alternative rewriting command `dsrewrite` (*depth strategic rewrite*) explores the strategy rewriting graph in depth. Its syntax coincides with that of the `srewrite` command except for the starting keyword, which can be abbreviated to `dsrew`.

`dsrewrite [`n`] in` ⟨*ModId*⟩ `:` ⟨*Term*⟩ `by` ⟨*StrategyExpression*⟩ `.`

Using this command in the previous example, we can observe some differences. The first solution is obtained sooner and some rewrites before the second. In absolute terms, both solutions are shown earlier and using fewer rewrites. The memory usage is reduced too.

```
Maude> dsrew [2] nil using solve .
dsrewrite [2] in BT-QUEENS : nil using solve .

Solution 1
rewrites: 15602 in 16ms cpu (17ms real) (942092 rewrites/second)
result NeList{Nat}: 1 5 8 6 3 7 2 4

Solution 2
rewrites: 20339 in 19ms cpu (21ms real) (1022728 rewrites/second)
result NeList{Nat}: 1 6 8 3 7 4 2 5
```

However, the disadvantage is that the depth-first exploration could get lost in an infinite branch before finding some reachable solutions, which would then be missed (see Section 10.5 for an example). The depth-first search order is determined by the order of the direct successors, which is summarized in the following precedences:

- Rule applications and `matchrew` matches are issued as in the `rewrite` and `match` commands, i.e., from outer to inner positions and from left to right. Rules are selected in their syntactical order within the module.

- In an alternative $\alpha \mid \beta$, the successors of α appear before the successors of β.
- In an iteration, leaving precedes continuing with another iteration.
- Strategy definitions are selected in the order they appear in the module. If the parameters match in different ways, these strategy executions are generated as in the `match` command.
- Rule and strategy condition fragments are processed from left to right and matches are generated as before. Rewriting condition fragments solutions are depth-first searched using these precedences.

10.5 Case study: logic programming

In this section, the strategy language is used to define the semantics of a logic programming language similar to Prolog [35]. Strategies will be used to discard failed proofs, to enforce the Prolog search strategy, and to implement advanced features like negation and cuts.

First, the syntax of the language is defined in a functional module:

```
fmod LP-SYNTAX is
  protecting NAT .
  protecting QID .

  sort Variable .
  op x{_} : Nat -> Variable [ctor] .

  sorts Term NvTerm .
  subsorts Qid < NvTerm < Term .
  subsort Variable < Term .
  op _[_] : Qid NeTermList -> NvTerm [ctor prec 30 gather (e &)] .

  sort NeTermList .
  subsort Term < NeTermList .
  op _,_ : NeTermList NeTermList -> NeTermList [ctor assoc] .

  sort Predicate .
  op _`(_`) : Qid NeTermList -> Predicate [ctor prec 30 gather (e &)] .

  sorts PredicateList NePredicateList .
  subsorts Predicate < NePredicateList < PredicateList .
  op nil : -> PredicateList [ctor] .
  op _,_ : PredicateList PredicateList -> PredicateList
                                    [ctor assoc prec 50 id: nil] .
  op _,_ : NePredicateList PredicateList -> NePredicateList [ditto] .
  op _,_ : PredicateList NePredicateList -> NePredicateList [ditto] .

  sort Clause .
  op _:-_ : Predicate PredicateList -> Clause [ctor prec 60] .

  sort Program .
  subsort Clause < Program .
  op nil : -> Program [ctor] .
  op _;_ : Program Program -> Program [ctor assoc prec 70 id: nil] .
endfm
```

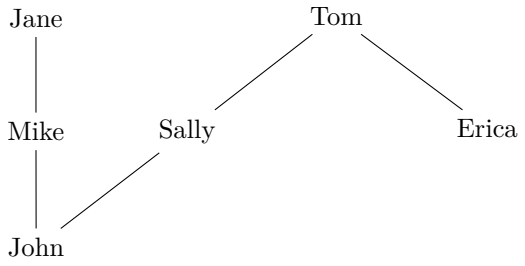


Figure 10.2: Family tree defined by the example predicates

The language distinguishes between *terms* and *predicates*: the first describe data while the second state facts about it. Terms are constants represented by quoted identifiers, variables of the form `x{`n`}`, and structured terms like `'f['a, 'b]`, built from symbols composed of a name and a list of terms between brackets. Predicates also take a non-empty list of terms but between parentheses. A *program* is a list of Horn clauses. They are composed of a predicate called *head*, the `:-` symbol, and a comma-separated list of predicates called *premises*. Clauses with an empty list of premises are called *facts*. For example, we can express familiar relations with predicates and people names as constants, then we can give some clauses to indicate how complex relationships derive from simpler ones:

```
ops kinship family : -> Program .

eq kinship =
  'sibling(x{1},x{2}) :- 'parent(x{3},x{1}), 'parent(x{3}, x{2}) ;
  'parent(x{1}, x{2}) :- 'father(x{1}, x{2}) ;
  'parent(x{1}, x{2}) :- 'mother(x{1}, x{2}) ;
  'relative(x{1},x{2}) :- 'parent(x{1},x{3}), 'parent(x{3},x{2}) ;
  'relative(x{1},x{2}) :- 'sibling(x{1},x{3}), 'relative(x{3},x{2}) .

eq family =
  'mother('jane, 'mike) :- nil ;
  'mother('sally, 'john) :- nil ;
  'father('tom, 'sally) :- nil ;
  'father('tom, 'erica) :- nil ;
  'father('mike, 'john) :- nil ;
  kinship .
```

According to the clauses above, Erica is sibling of Sally `'sibling('erica, 'sally)` because they both have Tom as parent, `'parent('tom, 'erica)` and `'parent('tom, 'sally)`. And this is true because he is the father of both, `'father('tom, 'erica)` and `'father('tom, 'sally)`.

The language interpreter is given a *query*, i.e., a list of predicates that may contain free variables, and it answers whether the conjunction of all of them can be satisfied and for which substitutions it does happen. The proof procedure consists in finding a clause whose head unifies with the predicate, and then trying to recursively prove their premises, while carrying the variable bindings during the process. Unification is implemented in a module `LP-UNIFICATION`, where substitutions are defined:

```
sort Binding .
op _->_ : Variable Term -> Binding [ctor prec 60] .
```

```
sort Substitution .
subsort Binding < Substitution .
op empty : -> Substitution [ctor] .
op _;_ : Substitution Substitution -> Substitution
          [ctor assoc comm prec 70 id: empty] .
```

The `unify` operator calculates the substitution that unifies two given predicates if possible, respecting the bindings from an initial substitution.

```
op unify : Predicate Predicate Substitution -> Substitution? .

sort Substitution? .
subsort Substitution < Substitution? .
op fail : -> Substitution? [ctor] .
```

The variables in a clause must be fresh each time it is applied, so they must be renamed. An operator does so helped by an index that is supposed to be greater that any variable index involved.

```
op rename : Clause Nat -> Clause .
```

Finally, a partial function `value` obtains the value of a variable in a given substitution:

```
op value : Variable Substitution ~> Term .
eq value(V, (V -> T) ; S) = T .
```

The language semantics is defined in a system module `LP-SEMANTICS`. The state of the interpreter is represented as a `Configuration`, and a rule modifies it while applying the clauses.

```
mod LP-SEMANTICS is
  protecting LP-UNIFICATION .

  sort Configuration .

  vars N1 N2 : Nat .                  vars P1 P2 P3 : Predicate .
  vars PL1 PL2 PL3 : PredicateList .  vars S1 S2 : Substitution .
  vars Pr Pr1 Pr2 : Program .

  op <_|_> : PredicateList Program -> Configuration [ctor] .
  eq < PL1 | Pr > = < last(PL1) | PL1 $ empty | Pr > .

  op <_|_$_|_> : Nat PredicateList Substitution Program -> Configuration [ctor] .

  crl [clause] : < N1 | P1, PL1 $ S1 | Pr1 ; P2 :- PL2 ; Pr2 >
      => < N2 | PL3, PL1 $ S2 | Pr1 ; P2 :- PL2 ; Pr2 >
      if P3 :- PL3 := rename(P2 :- PL2,N1)
      /\ S2 := unify(P1,P3,S1)
      /\ N2 := max(N1,last(P3 :- PL3)) .
endm
```

The interpreter state holds a predicate list of pending *goals*, and a substitution that accumulates the variable bindings from the previous execution. It also carries the program and maintains a renaming index greater than the index of any variable seen. The `clause` rule selects any program clause and tries to unify its head with the first predicate in the goal list. If it succeeds, the predicate is replaced by the clause premises and the current substitution is completed.

Additionally, it takes care of renaming the selected clause and updating the variable index counter.

A proof is successfully finished when a configuration of the form `< N | nil $ S | Pr >` is reached, i.e., when no pending goals remain, and then `S` is a substitution that makes the initial predicate hold.

Coming back to the family tree example, we can now ask questions like whether Jane and John are relatives, or of whom Tom is parent. As we want to know if a proof can be concluded for the initial predicate, `search` is the appropriate command. The value of `Pr` has been omitted in the following results for clarity.

```
Maude> search < 'relative('jane, 'john) | family > =>* < N | nil $ S | Pr > .
Solution 1 (state 11)
states: 12 rewrites: 1713
N:Nat --> 7
S:Substitution --> x{1} -> 'jane ; x{2} -> 'john ; x{3} -> x{5} ;
                  x{4} -> 'jane ; x{5} -> 'mike ; x{6} -> x{5} ; x{7} -> 'john

No more solutions.
states: 12 rewrites: 1967

Maude> search < 'parent('tom, x{1}) | family > =>* < N | nil $ S | Pr > .
Solution 1 (state 3)
states: 4 rewrites: 251
N:Nat --> 3
S:Substitution --> x{1} -> x{3} ; x{2} -> 'tom ; x{3} -> 'sally

Solution 2 (state 4)
states: 5 rewrites: 280
N:Nat --> 3
S:Substitution --> x{1} -> x{3} ; x{2} -> 'tom ; x{3} -> 'erica

No more solutions.
```

The `rewrite` command is not useful in this context because it simply explores a single rewriting path, thus a single proof path. This is clearly not enough to show multiple solutions, but it may also be insufficient to find a single one, as we can see running the first example with `rewrite`.

```
Maude> rew < 'relative('jane, 'john) | family > .
rewrites: 422
result Configuration: < 5 | 'father(x{4},x{5}),'parent(x{3},x{2}) $
x{1} -> 'jane ; x{2} -> 'john ; x{3} -> x{5} ; x{4} -> 'jane | family >
```

An admissible logic programming interpreter must consider all possible proof paths and be able to resume them when the execution arrives to a dead end. Strategies can take care of this. First, we define an auxiliary predicate `isSolution` to decide whether a given configuration is a solution.

```
mod LP-EXTRA is
  protecting LP-SEMANTICS .

  op isSolution : Configuration -> Bool .

  var N : Nat .          var S : Substitution .
  var Pr : Program .
```

```
  eq isSolution(< N | nil $ S | Pr >) = true .
  eq isSolution(Conf) = false [owise] .
endm
```

Now, we can define the strategy `solve-simple` that applies `clause` until a solution is found, and implicitly rejects any rewriting path that does not end in one. The exhaustive search of the `srewrite` command shows all reachable solutions for the initial predicate.

```
smod PROLOG is
  protecting LP-EXTRA .

  strat solve-simple @ Configuration .

  sd solve-simple := match Conf s.t. isSolution(Conf)
                     ? idle : (clause ; solve-simple) .
endsm
```

The above definition is identical to the backtracking scheme described in Section 10.3. In fact, the `BACKTRACKING` module parameterized by a convenient view can be used to define `solve-simple`. Even so, during this section, we will add some other elements to the strategy that are not compatible with that scheme. Now, the strategy can be applied to the previous examples:

```
Maude> srew < 'parent('tom, x{1}) | family > using solve-simple .

Solution 1
rewrites: 544
result Configuration: < 3 | nil $ x{1} -> x{3} ; x{2} -> 'tom ; x{3} -> 'sally
                        | (omitted) >

Solution 2
rewrites: 544
result Configuration: < 3 | nil $ x{1} -> x{3} ; x{2} -> 'tom ; x{3} -> 'erica
                        | (omitted) >

No more solutions.
rewrites: 544
```

The resulting configurations are not easily readable, because they include the full program, which has not changed and has been omitted here, and also substitutions containing meaningless variables that do not occur in the initial predicate. We propose to wrap solving strategies, like `solve-simple` and those we will define next, as a strategy `wsolve` that additionally cleans and presents the results in a more readable form. This is defined in the strategy module `PL-SIMPLIFIER`, which imports some required functions and rules from the system module `PL-SIMPLIFIER-BASE`. Since this wrapper is intended to be used for any solving strategy we might define, `PL-SIMPLIFIER` is a strategy module parameterized by a new theory `INTERPRETER` that strategies like `solve-simple` should satisfy.

```
sth INTERPRETER is
  protecting LP-SEMANTICS .
  strat solve @ Configuration .
endsth

view Simple from INTERPRETER to PROLOG is
  strat solve to solve-simple .
endv
```

```
smod PL-SIMPLIFIER{X :: INTERPRETER} is
  protecting PL-SIMPLIFIER-BASE .
  strat wsolve @ Configuration .
  var Conf : Configuration .
  var VS : VarSet .
  sd wsolve := matchrew Conf s.t. VS := occurs(Conf)
               by Conf using (solve ; solution[VS <- VS]) .
endsm
```

The strategy `wsolve` records the variables that occur in the initial configuration predicate, then executes the `solve` strategy, and finally applies the `solution` rule with the initial variable set. This rule restricts the substitution to the variables in the given set, after resolving them by transitivity. This procedure is defined in the `LP-SIMPLIFICATION-BASE` module, which is only partially reproduced here because it has no interest regarding strategies.

```
mod PL-SIMPLIFIER-BASE is
  extending LP-SEMANTICS .
  sort VarSet .
  subsort Variable < VarSet .
  op empty : -> VarSet .
  op _;_ : VarSet VarSet -> VarSet [ctor assoc comm id: empty] .

  *** [...]

  op occurs : Configuration -> VarSet .
  op simplify : Substitution VarSet -> Substitution .
  op solution : Substitution -> Configuration [ctor format (g! o)] .
  rl [solution] : < N | nil $ S | Pr > => solution(simplify(S, VS)) [nonexec] .
endm
```

Instantiating the module, we obtain a wrapped strategy that additionally pretty prints the solution. The same will be done for the other solving strategies defined in the rest of the section.

```
smod PROLOG-MAIN is
  protecting PL-SIMPLIFIER{Simple} * (strat wsolve to wsolve-simple) .
endsm
```

We can also observe that the order in which solutions appear depends on the way the rewriting tree is explored. With the `dsrewrite` command the results will appear in the same order as in Prolog, because both explore the derivation tree in depth. However, the `srewrite` command will often obtain shallower solutions first.

```
Maude> dsrew < 'p(x{1}) | 'p(x{1}) :- 'q(x{1}) ; 'p('a) :- nil ; 'q('b) :- nil >
        using wsolve-simple .

Solution 1
rewrites: 82
result Configuration: solution(x{1} -> 'b)

Solution 2
rewrites: 111
result Configuration: solution(x{1} -> 'a)

No more solutions.
rewrites: 117
```

```
Maude> srew < 'p(x{1}) | 'p(x{1}) :- 'q(x{1}) ; 'p('a) :- nil ; 'q('b) :- nil >
        using wsolve-simple .

Solution 1
rewrites: 105
result Configuration: solution(x{1} -> 'a)

Solution 2
rewrites: 117
result Configuration: solution(x{1} -> 'b)

No more solutions.
rewrites: 95
```

However, the benefit of using `srewrite` is that all reachable solutions are shown. In Prolog and with `dsrewrite` some of them may be hidden by a non-terminating branch.

```
Maude> dsrew < 'p(x{1}) | 'p(x{1}) :- 'p(x{1}) ; 'p('a) :- nil >
        using wsolve-simple .

Debug(1)> abort . *** non-terminating

Maude> srew < 'p(x{1}) | 'p(x{1}) :- 'p(x{1}) ; 'p('a) :- nil >
        using wsolve-simple .

Solution 1
rewrites: 109 in 0ms cpu (3ms real) (~ rewrites/second)
result Configuration: solution(x{1} -> 'a)

Debug(1)> abort . *** non-terminating
```

10.5.1 Negation as failure

In logic programming, the concept of negation is somewhat subtle. Facts and predicates express positive knowledge, so the alternatives are explicitly asserting what is false, or assuming that any predicate that cannot be derived from the considered facts is false. The last approach is known as *negation as failure*: the negation of a predicate holds if the predicate cannot be proved, no matter the values its variables take. This cannot be expressed with Horn clauses but can be easily implemented using strategies and an extra rewriting rule, added to `LP-EXTRA`. Negation is represented as a normal predicate named `\+`, whose argument is written as a term.

```
mod LP-EXTRA+NEG is
  extending LP-EXTRA .

  var N : Nat .                        var Q : Qid .
  var NeTL : NeTermList .              var PL : PredicateList .
  var S : Substitution .               var Pr : Program .
  var Conf : Configuration .

  crl [negation] : < N | '\+(Q[NeTL]), PL $ S | Pr > => < N | PL $ S | Pr >
    if < N | Q(NeTL) $ S | Pr > => Conf .
endm
```

The `negation` rule only removes the negation predicate from the goal list if its rewriting condition holds. By its own semantics, the negation never binds variables, so the substitution remains unchanged. The initial term of the rewriting condition contains the negated predicate as its only goal. Whether this term can be rewritten to a solution configuration determines whether the negated predicate can be satisfied. Hence, we need to control the condition with a strategy that fails whenever that happens.

```
smod PROLOG+NEG is
  protecting LP-EXTRA+NEG .

  strat solve-neg @ Configuration .

  var Conf : Configuration .

  sd solve-neg := match Conf s.t. isSolution(Conf) ? idle :
                  ((clause | negation{not(solve-neg)}) ; solve-neg) .
endsm
```

The strategy is similar to the original `solve-simple` strategy, but the `negation` rule can be applied when a negated predicate is on top of the goal list. `not(solve-neg)` fails if `solve-neg` finds a solution for the negated predicate. Otherwise, it behaves like `idle`, triggering the rule application. Thus, it is a suitable strategy for the rewriting condition.

We can illustrate the negation feature using the family tree example. A predicate `'no-children` claims that someone does not have descendants:

```
Maude> srew < 'no-children('erica) | family ;
          'no-children(x{1}) :- '\+('parent[x{1}, x{2}]) > using wsolve-neg .

Solution 1
rewrites: 887
result Configuration: solution(empty)

No more solutions.
rewrites: 887

Maude> srew < 'no-children('mike) | family ;
       'no-children(x{1}) :- '\+('parent[x{1}, x{2}]) > using solve-neg .

No solution.
rewrites: 894
```

The second predicate does not succeed since Mike is the father of John. Similarly, we can define `'orphan(x{1}) := '\+('parent[x{2}, x{1}])`.

10.5.2 Cuts

Prolog includes a more powerful (and more controversial) control feature called *cut*. Intuitively, a cut `!` is a goal that always succeeds and establishes a wall preventing backtracking left-wise. It can appear among the premises of a clause, and it triggers two effects when it is reached:

1. Locally to the clause, the first substitution produced to prove the predicates to the left of the cut is definitive, and it cannot be reconsidered.

2. Only the current clause can be used to prove the predicate to which the clause was applied. In other words, no other clause for this predicate can be used if the current clause fails

or more solutions are required.

Up to now, the order in which the solutions are found was less important, specially when using the `srew` command. However, for cut to be well-defined and coincide with its Prolog semantics, we need to respect its order strictly:

1. Inside a clause, the premises are proven from left to right.
2. To prove each goal, the program clauses are used in order, starting from the first one.

Keeping all this in mind, the cut-aware semantics is defined by means of two strategies `solve` and `solveOne`. The first one calculates all reachable solutions in no particular order, while the second calculates only the first solution according to the order described above. The ordered trial of the program clauses is conducted by two recursive strategies `clauseWalk` and `clauseWalkOne` receiving a program as argument. Before describing them, the `LP-SYNTAX` module should be extended with the cut symbol.

```
fmod LP-SYNTAX+CUT is
  extending LP-SYNTAX .

  sorts CfPredicateList NeCfPredicateList .
  subsorts Predicate < NeCfPredicateList
                   < CfPredicateList NePredicateList < PredicateList .

  op ! : -> NePredicateList [ctor] .

  mb nil : CfPredicateList .
  op _,_ : NeCfPredicateList NeCfPredicateList -> NeCfPredicateList [ditto] .
  op _,_ : CfPredicateList CfPredicateList -> CfPredicateList [ditto] .
  op _,_ : NeCfPredicateList CfPredicateList -> NeCfPredicateList [ditto] .
  op _,_ : CfPredicateList NeCfPredicateList -> NeCfPredicateList [ditto] .
  op _,_ : PredicateList  CfPredicateList -> PredicateList [ditto] .
  op _,_ : CfPredicateList PredicateList  -> PredicateList [ditto] .
endfm
```

The `CfPredicateList` sort represents cut-free predicate lists. These can be treated as usual, while clauses and goal lists containing cuts will be treated differently. The four strategies mentioned before are defined in the `PROLOG+CUT` module:

```
smod PROLOG+CUT is
  protecting LP-EXTRA+CUT .

  var  Conf   : Configuration .
  var  N      : NAt .
  var  CfPL   : CfPredicateList .
  var  P      : Predicate .
  vars Pr Pr2 : Program .
  vars PL PL2 : PredicateList .
  var  S      : Substitution .

  strats solve solveOne @ Configuration .

  sd solve := match Conf s.t. isSolution(Conf) or-else (
     matchrew Conf s.t. < N | CfPL $ S | Pr > := Conf
         by Conf using clauseWalk(Pr)
     | matchrew Conf s.t. < N | CfPL, !, PL $ S | Pr > := Conf
```

```
            by Conf using (cut{solveOne} ; solve)
  ) .

  sd solveOne := match Conf s.t. isSolution(Conf) or-else (
      matchrew Conf s.t. < N | CfPL $ S | Pr > := Conf
          by Conf using clauseWalkOne(Pr)
      | matchrew Conf s.t. < N | CfPL, !, PL $ S | Pr > := Conf
            by Conf using (cut{solveOne} ; solveOne)
  ) .

  strats clauseWalk clauseWalkOne : Program @ Configuration .

  sd clauseWalk(nil) := fail .
  sd clauseWalk(P :- PL2 ; Pr) := (clause[Pr2 <- Pr] ; solve) | clauseWalk(Pr) .
  sd clauseWalk(P :- CfPL, !, PL ; Pr) := (clause[Pr2 <- Pr] ; cut{solveOne})
      ? solve : clauseWalk(Pr) .

  sd clauseWalkOne(nil) := fail .
  sd clauseWalkOne(P :- CfPL ; Pr) := (clause[Pr2 <- Pr] ; solveOne)
      or-else clauseWalkOne(Pr) .
  sd clauseWalkOne(P :- CfPL, !, PL ; Pr) := (clause[Pr2 <- Pr] ; cut{solveOne})
      ? solveOne : clauseWalkOne(Pr) .
endsm
```

As in the previous strategies, `solve` stops when the subject term is a solution. Otherwise, two mutually exclusive branches are available. The first branch deals with cut-free goal lists, and simply invokes `clauseWalk` to attempt the program clauses in order. The second branch deals with cuts thanks to a new conditional rule `cut` defined in `LP-EXTRA`.

```
mod LP-EXTRA+CUT is
  extending LP-SYNTAX+CUT .
  extending LP-SEMANTICS+CALL .
  extending LP-EXTRA .

  vars N1 N2 : Nat .       var CfPL : CfPredicateList .
  var PL : PredicateList . vars S1 S2 : Substitution .
  var Pr : Program .

  crl [cut] : < N1 | CfPL, !, PL $ S1 | Pr > => < N2 | PL $ S2 | Pr >
    if < N1 | CfPL $ S1 | Pr > => < N2 | nil $ S2 | Pr > .
endm
```

This rule isolates the goal list to the left of the cut, which is cut-free. If a solution can be found for it, in the main configuration the left predicate list and the first cut are both removed, and the substitution is updated according to the result of the rewriting condition. Since cut disallows reconsidering the substitution that arrives to it, we are only interested in a single solution for the rewriting condition. This is achieved by controlling it with `solveOne`, which produces one solution at most. Notice that we are managing the cuts by anticipating them, and it works regardless of the way the strategy search tree is explored, and without explicit mention of backtracking. However, this only ensures the local effect of the cut; the second effect is guaranteed by `clauseWalk`. The recursive definition of `solveOne` only differs from `solve` in that `clauseWalkOne` is called instead of `clauseWalk`. The reason why only a solution is found by `solveOne` is because of `clauseWalkOne`, which is explained below.

`clauseWalk` is a recursive strategy that walks through the program clauses and tries them in

order. Setting `Pr2` to the argument's `Pr`, makes the application `clause[Pr2 <- Pr]` deterministic and ensures the current clause in the strategy arguments is used. If the clause succeeds, the execution continues with a recursive call to `solve`. No matter whether all this fails or not, the next clause is also tried to possibly find other solutions. `clauseWalkOne` differs only in that the disjuntive combinator `|` is replaced by an `or-else`. This ensures that only a single solution is found and that it is the first according to the order described before. The second effect of cut is guaranteed by the third definition of both `clauseWalkOne` and `clauseWalk`, that are dedicated to clauses with at least one cut in their premises. First, they apply the current clause using the `clause` rule. If the rule can be applied, its premises are transferred to the configuration goal list, where the cut is managed by `cut{solveOne}` as we explained above. Notice that the cut is only reached if `cut{solveOne}` succeeds. In that case, we continue with a recursive call to `solve`, ignoring the rest of clauses in `Pr` for the current predicate, as the second effect of cut requires. Otherwise, we continue with the next clause because the cut has not been triggered.

The examples in Section 10.5.1 can also be expressed by means of cuts.

```
Maude> srew < 'no-children('erica) | family ;
             'no-children(x{1}) :- 'parent(x{1}, x{2}), !, 'fail(x{1}) ;
             'no-children(x{1}) :- nil > using wsolve .

Solution 1
rewrites: 789
result Configuration: solution(empty)

No more solutions.
rewrites: 789
```

where `fail` is a predicate that always fails. This usage pattern of cut imitates the typical definition of the negation `\+` predicate. In fact, this predicate can be defined inside the language if completed by an extra meta-predicate named `call`, which converts its argument from a term to a predicate, as the `negation` rule implicitly did. It is implemented by means of an equation than reduces the head of the goal list:

```
fmod LP-SEMANTICS+CALL is
  protecting LP-SEMANTICS .

  var N : Nat .                          var Q : Qid .
  var NeTL : NeTermList .                var PL : PredicateList .
  var S : Substitution .                 var Pr : Program .
  var V : Variable .

  eq [call] : < N | 'call(Q[NeTL]), PL $ S | Pr >
            = < N | Q(NeTL), PL $ S | Pr > .

  ceq [call] : < N | 'call(V), PL $ S | Pr >
             = < N | Q(NeTL), PL $ S | Pr > if Q[NeTL] := value(V, S) .
endfm
```

Negation is then defined:

```
eq negation-bycut := '\+(x{1}) :- 'call(x{1}), !, 'fail(x{1}) ;
                     '\+(x{1}) :- nil .
```

This is a negation as failure, because whenever the `\+` predicate argument can be proven, the cut will be reached. Then, any other alternative proof for the negation will be discarded, in

particular the use of the second clause. Since `fail` always fails, the negation will fail completely. On the contrary, when the argument predicate does not produce any result, the cut will not be reached and the second clause will be used, thus proving the negation of the predicate.

```
Maude> srew < 'no-children('erica) | family ; negation-bycut ;
             'no-children(x{1}) :- '\+('parent[x{1}, x{2}]) > using wsolve .

Solution 1
rewrites: 1028
result Configuration: solution(empty)

No more solutions.
rewrites: 1028

Maude> srew < 'no-children('mike) | family ; negation-bycut ;
             'no-children(x{1}) :- '\+('parent[x{1}, x{2}]) > using wsolve .

No solution.
rewrites: 736
```

Chapter 11

Model Checking Invariants Through Search

A rewrite theory, specified in Maude as a system module, provides an executable mathematical model of a concurrent system. We can use the Maude specification to *simulate* the concurrent system so specified. But we can do more. Under appropriate conditions we can *check* that our mathematical model satisfies some important properties, or obtain a useful counterexample showing that the property in question is violated. This kind of *model-checking analysis* can be quite general. Chapter 12 will explain how, under appropriate finite reachability assumptions, we can model check any linear time temporal logic (LTL) property of a system specified in Maude as a system module. This chapter focuses on a simpler, yet very useful, model-checking capability, namely, the model checking of *invariants*, which can be accomplished just by using the `search` command.

11.1 Invariants

Invariants are the most common and useful *safety properties*, that is, properties stating that something bad should never happen. Given a transition system and an initial state s_0, an *invariant* I is a predicate defining a subset of states meeting two properties:

- it contains s_0, and
- it contains any state reachable from s_0 through a finite number of transitions.

Therefore, an invariant is a predicate defining a set of states that contains all the states reachable from s_0. If an invariant holds, that is, if it is truly an invariant satisfying the two properties above, then we know that something "bad" can never happen, namely, the negation $\neg I$ of the invariant is impossible. In other words, we view $\neg I$ as a bad property that should never happen, and I as a good property we want to ensure.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified in Maude as a system module, we can define an invariant `I`, yielding a decidable set of states, by:

1. choosing a given kind `k` in Σ as the kind of states and an initial state `init` in it; and
2. specifying an *equationally-defined Boolean condition* `I`, so that the invariant holds of a state if and only if such a state satisfies the condition `I`.

The transition system implicit in this setting is one in which a one-step transition between two states, that is, between two elements $[t], [t'] \in T_{\Sigma/E,\mathtt{k}}$, exists if and only if there is a representative $t_0 \in [t]$ and a one-step rewrite with the rules R, $t_0 \longrightarrow^1 t'_0$, such that $t'_0 \in [t']$. We introduce the notation

$$\mathcal{R}, \mathtt{init} \models \Box \mathtt{I}$$

to state that the transition system associated with $\mathcal{R}$ with state kind `k` and initial state `init` satisfies the invariant `I`.

11.2 Model checking of invariants

The key question now is: how can we automatically *verify* that an invariant `I` holds? The answer is very simple. Assuming that $\mathcal{R} = (\Sigma, E, \phi, R)$ satisfies reasonable executability conditions, namely, that E and R are finite sets, (Σ, E) is ground Church-Rosser and terminating,[1] and the rules R are ground coherent with E, and assuming, further, that all the conditions for rules in R are purely equational, then I holds if and only if the search command

```
search init =>* x:k such that I(x:k) =/= true .
```

has no solutions. Indeed, having no solutions exactly means that on `init`, and on all states reachable from it, the predicate `I` evaluates to `true`, that is, that `I` is an invariant. Assuming that `I` has been fully defined in all cases (which is always easy with the `owise` feature, described in Section 4.5.4), so that it always evaluates to either `true` or `false`, we could instead give the command

```
search init =>* x:k such that not I(x:k) .
```

Consider, for example, a simple clock that marks the hours of the day. Such a clock can be specified with the system module

```
mod SIMPLE-CLOCK is
  protecting INT .
  sort Clock .
  op clock : Int -> Clock [ctor] .
  var T : Int .
  rl clock(T) => clock((T + 1) rem 24) .
endm
```

A natural initial state is the state `clock(0)`. Note that, in principle, the clock could be in an infinite number of states, such as `clock(7633157)` or `clock(-33457129)`. The point, however, is that if our initial state is `clock(0)`, then only states `clock(T)` with times `T` such that `0 <= T < 24` can be reached. This suggests making the predicate `0 <= T < 24` an invariant of our clock system.

Using simple linear arithmetic reasoning, we can express the negation of such an invariant as the predicate `(T < 0) or (T >= 24)`; thus, we can automatically verify that our simple clock satisfies the invariant by giving the command:

```
Maude> search in SIMPLE-CLOCK : clock(0) =>* clock(T)
        such that T < 0 or T >= 24 .

No solution.
states: 24 rewrites: 216 in 0ms cpu (2ms real) (~ rews/sec)
```

[1] As usual, the ground Church-Rosser and termination assumptions should be understood *modulo* any axioms $A \subseteq E$ which in Maude are declared as equational attributes of operators.

If, as it is the case in this clock example, the number of states reachable from the initial state is *finite*, then search commands of this kind provide a *decision procedure* for the satisfaction of invariants. That is, in finite time Maude will either find no solutions to a search for a state violating the invariant, or will find a state violating the invariant together with a sequence of rewrites from the initial state to it, that is, a counterexample.

But what if the number of states reachable from the initial state is *infinite*? In such a case, *if* the invariant `I` is violated, the `search` command will terminate in finite time yielding a counterexample. Termination is guaranteed by the breadth-first nature of the search. The point is that such a counterexample is a *reachable* state; therefore, there is a finite sequence of rewrites from the initial state to such a violating state. Since there is only a finite number of rules R, and therefore a finite number of ways that each state can be rewritten, even though the number of reachable states is infinite, the number of states reachable from the initial state by a sequence of rewrites of length less than a given bound is always finite. This bounded subset is always explored in finite time by the `search` command. This means that, for systems where the set of reachable states is infinite, search becomes a *semi-decision procedure* for detecting the *violation* of an invariant. That is, if the invariant is violated, we are guaranteed to get a counterexample; but, if it is not violated, we will search forever, never finding it.

We can illustrate the semi-decision procedure nature of search for the verification of invariant failures with a simple infinite-state example of processes and resources. This example has some similarities with the classical dining philosophers problem, but it is on the one hand simpler (processes and resources have no identities or topology), and on the other hand more complex, since the number of processes and resources can grow dynamically in an unbounded manner.

```
mod PROCS-RESOURCES is
  sorts State Resources Process .
  subsorts Process Resources < State .
  ops res null : -> Resources [ctor] .
  op p : Resources -> Process [ctor] .
  op __ : Resources Resources -> Resources
       [ctor assoc comm id: null] .
  op __ : State State -> State [ctor ditto] .

  rl [acq1] : p(null) res => p(res) .
  rl [acq2] : p(res) res => p(res res) .
  rl [rel] : p(res res) => p(null) res res .
  rl [dupl] : p(null) res => p(null) res p(null) res .
endm
```

The state is a soup (multiset) of processes and resources. Each process needs to acquire two resources. Originally, each process `p` contains the `null` state; but if a resource `res` is available, it can acquire it (rule `acq1`). If a second resource becomes available, it can also acquire it (rule `acq2`). After acquiring both resources and using them, the process can release them (rule `rel`). Furthermore, the number of processes and resources can grow in an unbounded manner by the duplication of each process-resource pair (rule `dupl`).

One invariant we might like to verify about this system is *deadlock freedom*. There are two ways to model check this property: one completely straightforward, and another requiring some extra work. The straightforward manner is to give the search command

```
Maude> search in PROCS-RESOURCES : res p(null) =>! X:State .

Solution 1 (state 1)
states: 3 rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res)
```

```
Solution 2 (state 5)
states: 9 rewrites: 9 in 0ms cpu (1ms real) (~ rews/sec)
X:State --> p(res) p(res)

Solution 3 (state 13)
states: 19 rewrites: 26 in 0ms cpu (3ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res)

Solution 4 (state 25)
states: 34 rewrites: 56 in 0ms cpu (4ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res)

Solution 5 (state 43)
states: 55 rewrites: 104 in 0ms cpu (23ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res) p(res)

......

Solution 20 (state 1649)
states: 1770 rewrites: 5640 in 20ms cpu (67ms real)
  (282000 rews/sec)
X:State --> p(res) p(res) p(res) p(res) p(res) p(res) p(res) p(res)
           p(res) p(res) p(res) p(res) p(res) p(res) p(res) p(res)
           p(res) p(res) p(res) p(res)
......
```

Maude will indeed continue printing all the solutions it finds. But since there is an infinite number of deadlock states, it may be preferable to specify in advance a bound on the number of solutions, giving, for example, a command like the following, that looks for at most 5 solutions.

```
Maude> search [5] in PROCS-RESOURCES : res p(null) =>! X:State .
```

The nice thing about model checking deadlock freedom this way is that there is no need to explicitly specify the invariant as a Boolean predicate. This is because the negation of the invariant is by definition the set of deadlock states, which is what the `search` command with the `=>!` qualification precisely looks for.

If one wishes, one can, with a little more work, perform an equivalent model checking of the same property by using an explicit enabledness predicate. Of course, this cannot be done in the original module, because such a predicate has not been defined, but this is easy enough to do:

```
mod PROCS-RESOURCES-ENABLED is
  protecting PROCS-RESOURCES .
  protecting BOOL .
  op enabled : State -> Bool .
  eq enabled(p(null) res X:State) = true .
  eq enabled(p(res) res X:State) = true .
  eq enabled(p(res res) X:State) = true .
  eq enabled(X:State) = false [owise] .
endm
```

One can then give the command

```
Maude> search [5] in PROCS-RESOURCES-ENABLED : res p(null)
        =>* X:State such that enabled(X:State) =/= true .
```

getting the following 5 solutions:

```
Solution 1 (state 1)
states: 2 rewrites: 4 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res)

Solution 2 (state 5)
states: 6 rewrites: 15 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res) p(res)

Solution 3 (state 13)
states: 14 rewrites: 41 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res)

Solution 4 (state 25)
states: 26 rewrites: 87 in 0ms cpu (1ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res)

Solution 5 (state 43)
states: 44 rewrites: 160 in 0ms cpu (1ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res) p(res)
```

11.3 Bounded model checking of invariants

In cases where either the set of reachable states is infinite, or it is finite but too large to be explored in reasonable time due to time and memory limitations, *bounded model checking* is an appealing formal analysis method. The idea of bounded model checking is that we model check a property, not for all reachable states, but only for those states reachable within a certain *depth bound*, that is, reachable by a sequence of transitions of bounded length. Of course, our analysis is not complete anymore, since we may fail to find a counterexample lying at greater depth. However, bounded model checking can be quite effective in finding counterexamples and it is a widely used procedure. Bounded model checking of invariants is supported in Maude by means of the *bounded search command*.

Consider the following specification of a readers-writers system.

```
mod READERS-WRITERS is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  sort Config .
  op <_,_> : Nat Nat -> Config [ctor] . --- readers/writers

  vars R W : Nat .

  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .
  rl < s(R), W > => < R, W > .
endm
```

A state is represented by a tuple `< R, W >` indicating the number `R` of readers and the number `W` of writers accessing a critical resource. Readers and writers can leave the resource at any

time, but writers can only gain access to it if nobody else is using it, and readers only if there are no writers.

Taking `< 0, 0 >` as the initial state, this simple system satisfies two important invariants, namely:

- *mutual exclusion*: readers and writers never access the resource simultaneously: only readers or only writers can do so at any given time.
- *one writer*: at most one writer will be able to access the resource at any given time.

We could try model checking these two invariants in two different ways:

- we can represent the invariants *implicitly* by representing their negations through *patterns*; or
- we can represent them *explicitly* as Boolean predicates.

To model check our two invariants using an implicit representation we could use the commands

```
Maude> search < 0, 0 > =>* < s(N:Nat), s(M:Nat) > .

Maude> search < 0, 0 > =>* < N:Nat, s(s(M:Nat)) > .
```

since the negation of the first invariant corresponds to the simultaneous presence of both readers and writers, which is exactly captured by the first pattern `< s(N:Nat), s(M:Nat) >`; whereas the negation of the fact that zero or at most one writer should be present at any given time is exactly captured by the second pattern `< N:Nat, s(s(M:Nat)) >`.

The problem with the above two search comands is that, since the number or readers allowed is unbounded, the set of reachable states is infinite and the commands never terminate. We can instead perform bounded model checking of these two invariants by giving a depth bound, for example 10^5, with the commands:

```
Maude> search [1, 100000] in READERS-WRITERS :
        < 0, 0 > =>* < s(N:Nat), s(M:Nat) > .

No solution.
states: 100002 rewrites: 200001 in 312460ms cpu (636669ms real)
  (640 rews/sec)

Maude> search [1, 100000] in READERS-WRITERS :
        < 0, 0 > =>* < N:Nat, s(s(M:Nat)) > .

No solution.
states: 100002 rewrites: 200001 in 70600ms cpu (623434ms real)
  (2832 rews/sec)
```

As the reader can observe, these computations take quite a long time. Notice that the terms appearing during the search grow very quickly. A very simple way of improving performance in this case is using the `iter` attribute for the `s` operator.

```
op s : Nat -> Nat [ctor iter] .
```

Then, we obtain a much better performance:

```
Maude> search [1, 100000] in READERS-WRITERS :
        < 0, 0 > =>* < s(N:Nat), s(M:Nat) > .
```

```
No solution.
states: 100002 rewrites: 200001 in 610ms cpu (1298ms real)
  (327870 rews/sec)

Maude> search [1, 100000] in READERS-WRITERS :
        < 0, 0 > =>* < N:Nat, s(s(M:Nat)) > .

No solution.
states: 100002 rewrites: 200001 in 400ms cpu (1191ms real)
  (500002 rews/sec)
```

In the following section we will use some formal tools for checking properties about the `READERS-WRITERS` module. Since some of these tools cannot handle the `iter` attribute, we use the module as shown above.

11.4 Verifying infinite-state systems through abstractions

The bounded model checking of our two invariants for the readers and writers example up to depth 10^6 greatly increases our confidence that the invariants hold, but, as already mentioned, bounded model checking is an incomplete procedure that falls short of being a proof: a counterexample at greater depth could exist.

Can we actually fully verify such invariants in an automatic way? One possible method is to verify the invariants through search not on the original infinite-state system, but on a finite-state *abstraction* of it, that is, on an appropriate quotient of the original system whose set of reachable states is finite. The paper [113] describes a simple method for, given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, defining an abstraction $\mathcal{A}$ of it: *just add equations*. That is, we can define our abstract theory as a rewrite theory $\mathcal{A} = (\Sigma, E \cup G, \phi, R)$, with G a set of extra equations powerful enough to collapse the infinite set of reachable states into a finite set. This method can be used (with an additional deadlock-freedom requirement) to verify not just invariants, but in fact any LTL formula (see [113] and Section 13.4 of [29]).

Of course, the equations G we add cannot be arbitrary. First of all, they should respect all the necessary executability assumptions, so that in $\mathcal{A} = (\Sigma, E \cup G, \phi, R)$ the equations $E \cup G$ should be ground Church-Rosser and terminating,[2] and the rules R should be ground coherent with $E \cup G$. Furthermore, the equations G should be *invariant-preserving* with respect to the invariants that we want to model check; that is, for any such invariant `I` and for any two ground terms t, t' denoting states such that $t =_{E \cup G} t'$, it should be the case that $E \vdash \mathtt{I}(t) = \mathtt{I}(t')$.

A first key observation is that, if both $\mathcal{R}$ and $\mathcal{A}$ *protect* the sort `Bool`, that is, the only canonical forms of that sort are `true` and `false`, and $\mathtt{true} \neq \mathtt{false}$, then the equations G are invariant-preserving. A second key observation is that we may be able to automatically check that a module protects the sort `Bool` by:

1. checking that it has no equations with `true` or `false` in the lefthand side;
2. checking that it is ground confluent and sort-decreasing with the Church-Rosser Checker (CRC) tool;
3. checking that it is terminating with the Maude Termination Tool (MTT); and
4. checking that it is sufficiently complete with the Sufficient Completeness Checker (SCC) tool.

[2] Again, possibly *modulo* equational attributes $A \subseteq E \cup G$.

Indeed, since `true` and `false` are the only constructors of sort `Bool`, (4) ensures the "no junk" part of protection, whereas (1)–(3) ensure the "no confusion," `true` $\neq$ `false` part.

For invariant verification, the key property that an abstraction meeting these requirements satisfies is that, if `I` is one of the invariants preserved by G, then we have the implication

$$\mathcal{A}, \texttt{init} \models \Box \texttt{I} \quad \Longrightarrow \quad \mathcal{R}, \texttt{init} \models \Box \texttt{I}$$

Therefore, if we can verify the invariant on $\mathcal{A}$, we are sure that it also holds on $\mathcal{R}$. However, the fact that we find a counterexample in $\mathcal{A}$ does not necessarily mean that a counterexample exists for $\mathcal{R}$: it could be a spurious counterexample, caused by $\mathcal{A}$ being too coarse of an abstraction, and therefore having no counterpart in $\mathcal{R}$. In such cases, one possible approach is to refine our abstraction by imposing fewer equations.

We can illustrate these ideas by defining an abstraction of our readers-writers system. In order to check that the equations in our abstraction preserve the invariants, we need an *explicit* representation of those invariants. Since at present the CRC and MTT tools do not handle predefined modules like `BOOL`, we use instead a sort `NewBool`.

```
mod READERS-WRITERS-PREDS is
  protecting READERS-WRITERS .
  sort NewBool .
  ops tt ff : -> NewBool [ctor] .
  ops mutex one-writer : Config -> NewBool [frozen] .
  eq mutex(< s(N:Nat), s(M:Nat) >) = ff .
  eq mutex(< 0, N:Nat >) = tt .
  eq mutex(< N:Nat, 0 >) = tt .
  eq one-writer(< N:Nat, s(s(M:Nat)) >) = ff .
  eq one-writer(< N:Nat, 0 >) = tt .
  eq one-writer(< N:Nat, s(0) >) = tt .
endm
```

We can now define our abstraction, in which all the states having more than one reader and no writers are identified with the state having exactly one reader and no writer.

```
mod READERS-WRITERS-ABS is
  including READERS-WRITERS-PREDS .
  including READERS-WRITERS .
  eq < s(s(N:Nat)), 0 > = < s(0), 0 > .
endm
```

where the second `including` importation is needed because the `READERS-WRITERS` module is not protected, but would be assumed protected by default (because it is protected in `READERS-WRITERS-PREDS`) unless we explicitly declare it in `including` mode (see Section 6.1.3).

In order to check both the executability and the invariant-preservation properties of this abstraction, since we have no equations with either `tt` or `ff` in their lefthand side, we now just need to check:

1. that the equations in both `READERS-WRITERS-PREDS` and `READERS-WRITERS-ABS` are ground confluent, sort-decreasing, and terminating;

2. that the equations in both `READERS-WRITERS-PREDS` and `READERS-WRITERS-ABS` are sufficiently complete; and

3. that the rules in both `READERS-WRITERS-PREDS` and `READERS-WRITERS-ABS` are ground coherent with respect to their equations.

Regarding termination, since the equations of READERS-WRITERS-ABS contain those of the module READERS-WRITERS-PREDS, it is enough to check the termination of the equations in the former. The MTT tool, using the AProVE termination tool, checks this automatically.

Once the READERS-WRITERS-ABS and READERS-WRITERS-PREDS modules are available in Full Maude (see Section 21.1), we can check confluence of the equations by invoking the CRC as follows:

```
Maude> (check Church-Rosser READERS-WRITERS-PREDS .)
Church-Rosser checking of READERS-WRITERS-PREDS
Checking solution:
  All critical pairs have been joined. The specification is
    locally-confluent.
The specification is sort-decreasing.

Maude> (check Church-Rosser READERS-WRITERS-ABS .)
Church-Rosser checking of READERS-WRITERS-ABS
Checking solution:
  All critical pairs have been joined. The specification is
    locally-confluent.
The specification is sort-decreasing.
```

which finishes task (1).

Regarding (2), the SCC tool gives us:

```
Maude> (scc READERS-WRITERS-PREDS .)
Checking sufficient completeness of READERS-WRITERS-PREDS ...
Success: READERS-WRITERS-PREDS is sufficiently complete under the
  assumption that it is weakly-normalizing, confluent, and
  sort-decreasing.

Maude> (scc READERS-WRITERS-ABS .)
Checking sufficient completeness of READERS-WRITERS-ABS ...
Success: READERS-WRITERS-ABS is sufficiently complete under the
  assumption that it is weakly-normalizing, confluent, and
  sort-decreasing.
```

This leaves us with task (3), where the Coherence Checker (ChC) tool responds as follows:

```
Maude> (check coherence READERS-WRITERS-PREDS .)
Coherence checking of READERS-WRITERS-PREDS
Coherence checking solution:
  All critical pairs have been rewritten and all equations
    are non-constructor.
  The specification is coherent.

Maude> (check coherence READERS-WRITERS-ABS .)
Coherence checking of READERS-WRITERS-ABS
Coherence checking solution:
  The following critical pairs cannot be rewritten:
  cp < s(0), 0 > => < s(N*@:Nat), 0 > .
```

Of course, ground coherence means that all ground instances of this pair can be rewritten by a one-step rewrite up to canonical form by the equations. We can reason by cases and decompose this critical pair into two:

```
cp < s(0), 0 > => < s(0), 0 > .
cp < s(0), 0 > => < s(s(N:Nat)), 0 > .
```

Using the `reduce` command we can check that the canonical form of the term `< s(s(N:Nat)), 0 >` is `< s(0), 0 >`. Therefore, all we need to do is to check that `< s(0), 0 >` rewrites to *itself* (up to canonical form) in one step. We can do this check by giving the command:

```
Maude> search in READERS-WRITERS-ABS : < s(0), 0 > =>1 X:Config .

Solution 1 (state 0)
states: 1 rewrites: 2 in 0ms cpu (26ms real) (~ rews/sec)
X:Config --> < s(0), 0 >

Solution 2 (state 1)
states: 2 rewrites: 3 in 0ms cpu (124ms real) (~ rews/sec)
X:Config --> < 0, 0 >

No more solutions.
```

We have therefore completed all the checks (1)–(3) and can now automatically verify the two invariants on the abstract system, and therefore conclude that they hold in our original infinite-state readers-writers system, by executing the `search` commands:

```
Maude> search in READERS-WRITERS-ABS :
        < 0, 0 > =>* C:Config such that mutex(C:Config) = ff .

No solution.
states: 3 rewrites: 9 in 0ms cpu (0ms real) (~ rews/sec)

Maude> search in READERS-WRITERS-ABS :
        < 0, 0 > =>* C:Config such that one-writer(C:Config) = ff .

No solution.
states: 3 rewrites: 9 in 0ms cpu (0ms real) (~ rews/sec)
```

Chapter 12

LTL Model Checking

As pointed out in Section 1.4, given a Maude system module, we can distinguish two levels of specification:

- a *system specification* level, provided by the rewrite theory specified by that system module which defines the behavior of the system, and
- a *property specification* level, given by some property (or properties) φ that we want to state and prove about our module.

This chapter discusses how a specific property specification logic, *linear temporal logic* (LTL), and a decision procedure for it, *model checking*, can be used to prove properties when the set of states reachable from an initial state in a system module is finite. It also explains how this is supported in Maude by its `MODEL-CHECKER` module, and other related modules, including the `SAT-SOLVER` module that can be used to check both satisfiability of an LTL formula and LTL tautologies. These modules are found in the file `model-checker.maude`.

Temporal logic allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens). These properties are related to the *infinite behavior* of a system. There are different temporal logics [24]; we focus on linear temporal logic [96, 24], because of its intuitive appeal, widespread use, and well-developed proof methods and decision procedures.

12.1 LTL formulas and the `LTL` module

Given a set AP of *atomic propositions*, we define the formulas of the *propositional linear temporal logic* LTL(AP) inductively as follows:

- **True**: $\top \in \mathrm{LTL}(AP)$.
- **Atomic propositions**: If $p \in AP$, then $p \in \mathrm{LTL}(AP)$.
- **Next operator**: If $\varphi \in \mathrm{LTL}(AP)$, then $\bigcirc\varphi \in \mathrm{LTL}(AP)$.
- **Until operator**: If $\varphi, \psi \in \mathrm{LTL}(AP)$, then $\varphi \,\mathcal{U}\, \psi \in \mathrm{LTL}(AP)$.
- **Boolean connectives**: If $\varphi, \psi \in \mathrm{LTL}(AP)$, then the formulas $\neg\varphi$, and $\varphi \vee \psi$ are in LTL(AP).

Other LTL connectives can be defined in terms of the above minimal set of connectives as follows:

- Other Boolean connectives:
  - **False**: $\bot = \neg\top$
  - **Conjunction**: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
  - **Implication**: $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$.
- Other temporal operators:
  - **Eventually**: $\Diamond\varphi = \top\, \mathcal{U}\, \varphi$
  - **Henceforth**: $\Box\varphi = \neg\Diamond\neg\varphi$
  - **Release**: $\varphi\, \mathcal{R}\, \psi = \neg((\neg\varphi)\, \mathcal{U}\, (\neg\psi))$
  - **Unless**: $\varphi\, \mathcal{W}\, \psi = (\varphi\, \mathcal{U}\, \psi) \vee (\Box\varphi)$
  - **Leads-to**: $\varphi \rightsquigarrow \psi = \Box(\varphi \rightarrow (\Diamond\psi))$
  - **Strong implication**: $\varphi \Rightarrow \psi = \Box(\varphi \rightarrow \psi)$
  - **Strong equivalence**: $\varphi \Leftrightarrow \psi = \Box(\varphi \leftrightarrow \psi)$.

The LTL syntax, in a typewriter approximation of the above mathematical syntax, is supported in Maude by the following `LTL` functional module (in the file `model-checker.maude`).

```
fmod LTL is
  protecting BOOL .
  sorts Formula .

  *** primitive LTL operators
  ops True False : -> Formula [ctor format (g o)] .
  op ~_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _/\_ : Formula Formula -> Formula
      [comm ctor gather (E e) prec 55 format (d r o d)] .
  op _\/_ : Formula Formula -> Formula
      [comm ctor gather (E e) prec 59 format (d r o d)] .
  op O_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _U_ : Formula Formula -> Formula
      [ctor prec 63 format (d r o d)] .
  op _R_ : Formula Formula -> Formula
      [ctor prec 63 format (d r o d)] .

  *** defined LTL operators
  op _->_ : Formula Formula -> Formula
      [gather (e E) prec 65 format (d r o d)] .
  op _<->_ : Formula Formula -> Formula [prec 65 format (d r o d)] .
  op <>_ : Formula -> Formula [prec 53 format (r o d)] .
  op []_ : Formula -> Formula [prec 53 format (r d o d)] .
  op _W_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
  op _|->_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
  op _=>_ : Formula Formula -> Formula
      [gather (e E) prec 65 format (d r o d)] .
  op _<=>_ : Formula Formula -> Formula [prec 65 format (d r o d)] .

  vars f g : Formula .
```

```
  eq f -> g = ~ f \/ g .
  eq f <-> g = (f -> g) /\ (g -> f) .
  eq <> f = True U f .
  eq [] f = False R f .
  eq f W g = (f U g) \/ [] f .
  eq f |-> g = [](f -> (<> g)) .
  eq f => g = [] (f -> g) .
  eq f <=> g = [] (f <-> g) .

*** negative normal form
  eq ~ True = False .
  eq ~ False = True .
  eq ~ ~ f = f .
  eq ~ (f \/ g) = ~ f /\ ~ g .
  eq ~ (f /\ g) = ~ f \/ ~ g .
  eq ~ O f = O ~ f .
  eq ~ (f U g) = (~ f) R (~ g) .
  eq ~ (f R g) = (~ f) U (~ g) .
endfm
```

Note that, for the moment, no set AP of atomic propositions has been specified in the `LTL` module. Section 12.2 explains how such atomic propositions are defined for a given system module `M`, and Section 12.3 explains how they are added to the `LTL` module as a subsort `Prop` of `Formula` in the `MODEL-CHECKER` module.

Note that the nonconstructor connectives have been defined in terms of more basic constructor connectives in the first set of equations. But since there are good reasons to put an LTL formula in *negative normal form* by pushing the negations next to the atomic propositions (this is specified by the second set of equations) we need to consider also the *duals* of the basic connectives $\top$, $\bigcirc$, $\mathcal{U}$, and $\vee$ (respectively, `True`, `O_`, `_U_`, and `_\/_`) as constructors. That is, we need to also have as constructors the dual connectives: $\bot$, $\mathcal{R}$, and $\wedge$ (respectively, `False`, `_R_`, and `_/\_`). Note that $\bigcirc$ is self-dual.

12.2 Associating Kripke structures to rewrite theories

Since the models of temporal logic are Kripke structures, we need to explain how we can associate a Kripke structure to the rewrite theory specified by a Maude system module `M`.

Kripke structures are the natural models for propositional temporal logic. Essentially, a Kripke structure is a (total) *unlabeled transition system* to which we have added a collection of unary state predicates on its set of states.

A binary relation $R \subseteq A \times A$ on a set A is called *total* if and only if for each $a \in A$ there is at least one $a' \in A$ such that $(a, a') \in R$. If R is not total, it can be made total by defining $R^\bullet = R \cup \{(a, a) \in A^2 \mid \nexists a' \in A\ (a, a') \in R\}$.

A *Kripke structure* is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ such that A is a set, called the set of *states*, $\rightarrow_{\mathcal{A}}$ is a total binary relation on A, called the *transition relation*, and $L : A \longrightarrow \mathcal{P}(AP)$ is a function, called the *labeling function*, associating to each state $a \in A$ the set $L(a)$ of those *atomic propositions* in AP that *hold* in the state a.

The semantics of LTL(AP) is defined by means of a *satisfaction relation*

$$\mathcal{A}, a \models \varphi$$

between a Kripke structure $\mathcal{A}$ having AP as its atomic propositions, a state $a \in A$, and an LTL formula $\varphi \in \mathrm{LTL}(AP)$. Specifically, $\mathcal{A}, a \models \varphi$ holds if and only if for each path $\pi \in Path(\mathcal{A})_a$ the *path satisfaction relation*

$$\mathcal{A}, \pi \models \varphi$$

holds, where we define the set $Path(\mathcal{A})_a$ of *computation paths* starting at state a as the set of functions of the form $\pi : \mathbb{N} \longrightarrow A$ such that $\pi(0) = a$ and, for each $n \in \mathbb{N}$, we have $\pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$.

We can define the path satisfaction relation (for any path, beginning at any state) inductively as follows:

- We always have $\mathcal{A}, \pi \models_{LTL} \top$.
- For $p \in AP$,
$$\mathcal{A}, \pi \models_{LTL} p \quad \Leftrightarrow \quad p \in L(\pi(0)).$$
- For $\bigcirc\varphi \in \mathrm{LTL}(A)$,
$$\mathcal{A}, \pi \models_{LTL} \bigcirc\varphi \quad \Leftrightarrow \quad \mathcal{A}, s; \pi \models_{LTL} \varphi,$$
where $s : \mathbb{N} \longrightarrow \mathbb{N}$ is the successor function, and where $(s;\pi)(n) = \pi(s(n)) = \pi(n+1)$.
- For $\varphi \,\mathcal{U}\, \psi \in \mathrm{LTL}(\mathcal{A})$,
$$\mathcal{A}, \pi \models_{LTL} \varphi \,\mathcal{U}\, \psi \quad \Leftrightarrow$$
$$(\exists n \in \mathbb{N})$$
$$((\mathcal{A}, s^n; \pi \models_{LTL} \psi) \wedge ((\forall m \in \mathbb{N})\ m < n \ \Rightarrow \mathcal{A}, s^m; \pi \models_{LTL} \varphi)).$$
- For $\neg\varphi \in \mathrm{LTL}(AP)$,
$$\mathcal{A}, \pi \models_{LTL} \neg\varphi \quad \Leftrightarrow \quad \mathcal{A}, \pi \not\models_{LTL} \varphi.$$
- For $\varphi \vee \psi \in \mathrm{LTL}(AP)$,
$$\mathcal{A}, \pi \models_{LTL} \varphi \vee \psi \quad \Leftrightarrow \quad \mathcal{A}, \pi \models_{LTL} \varphi \ \text{ or } \ \mathcal{A}, \pi \models_{LTL} \psi.$$

How can we associate a Kripke structure to the rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by a Maude system module `M`? We just need to make explicit two things:

- the intended *kind* k of states in the signature Σ, and
- the relevant *state predicates*, that is, the relevant set AP of atomic propositions.

In general, the state predicates need not be part of the *system specification* and therefore they need not be specified in our system module `M`. They are typically part of the *property specification*. This is because the state predicates need not be related to the operational semantics of `M`: they are just certain *predicates* about the states of the system specified by `M` that are needed to specify some *properties*.

Therefore, after choosing a given kind, say `[Foo]`, in `M` as our kind for states we can specify the relevant state predicates in a module `M-PREDS` protecting `M` according to the following general pattern:

```
mod M-PREDS is
  protecting M .
  including SATISFACTION .
  subsort Foo < State .
  ...
endm
```

where the dots '...' indicate the part in which the syntax and semantics of the relevant state predicates are specified, as further explained in what follows.

The module SATISFACTION (contained in the model-checker.maude file) is very simple, and has the following specification:

```
fmod SATISFACTION is
  protecting BOOL .
  sorts State Prop .
  op _|=_ : State Prop -> Bool [frozen] .
endfm
```

where the sorts State and Prop are unspecified. However, by importing SATISFACTION into M-PREDS and giving the subsort declaration

```
subsort Foo < State .
```

all terms of sort Foo in M are also made terms of sort State. Note that we then have the kind identity [Foo] = [State].

The operator

```
op _|=_ : State Prop -> Bool [frozen] .
```

is crucial to define the semantics of the relevant state predicates in M-PREDS. Each such state predicate is declared as an operator of sort Prop. In standard LTL propositional logic, the set AP of atomic propositions is assumed to be a set of *constants*. In Maude, however, we can define *parametric* state predicates, that is, operators of sort Prop which need not be constants, but may have one or more sorts as parameter arguments. We then define the *semantics* of such state predicates (when the predicate holds) by appropriate equations.

We can illustrate all this by means of a simple mutual exclusion example. Suppose that our original system module M is the following module MUTEX, in which two processes, one named a and another named b, can be either waiting or in their critical section, and take turns accessing their critical section by passing each other a different *token* (either $ or *).

```
mod MUTEX is
  sorts Name Mode Proc Token Conf .
  subsorts Token Proc < Conf .
  op none : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: none] .
  ops a b : -> Name [ctor] .
  ops wait critical : -> Mode [ctor] .
  op [_,_] : Name Mode -> Proc [ctor] .
  ops * $ : -> Token [ctor] .
  rl [a-enter] : $ [a, wait] => [a, critical] .
  rl [b-enter] : * [b, wait] => [b, critical] .
  rl [a-exit] : [a, critical] => [a, wait] * .
  rl [b-exit] : [b, critical] => [b, wait] $ .
endm
```

Our obvious kind for states is the kind [Conf] of configurations. In order to state the desired safety and liveness properties we need state predicates telling us whether a process is waiting or is in its critical section. We can make these predicates *parametric* on the name of the process and define their semantics as follows:

```
mod MUTEX-PREDS is
  protecting MUTEX .
  including SATISFACTION .
```

```
  subsort Conf < State .
  op crit : Name -> Prop .
  op wait : Name -> Prop .
  var N : Name .
  var C : Conf .
  var P : Prop .
  eq [N, critical] C |= crit(N) = true .
  eq [N, wait] C |= wait(N) = true .
  eq C |= P = false [owise] .
endm
```

Note the equations, defining when each of the two parametric state predicates holds in a given state.

The above example illustrates a general method by which desired state predicates for a module `M` are defined in a *protecting* extension, say `M-PREDS`, of `M` which imports `SATISFACTION`. One specifies the desired states by choosing a sort in `M` and declaring it as a subsort of `State`. One then defines the syntax of the desired state predicates as operators of sort `Prop`, and defines their semantics by means of a set of equations that specify for what states a given state predicate evaluates to `true`. We assume that those equations, when added to those of `M`, are (ground) Church-Rosser and terminating, and, furthermore, that `M`'s equational part is *protected* when the new operators and equations defining the state predicates are added.

Since we should *protect* `BOOL`, it is important to make sure that satisfaction of state predicates is *fully defined*. This can be checked with Maude's Sufficient Completeness Checker (SCC) tool. This means that we should give equations for when the predicates are `true` and when they are `false`. In practice, however, this can often be reduced to specifying *when a predicate is true* by means of (possibly conditional) equations of the general form,

$$t \models p(v_1, \ldots, v_n) = \mathit{true} \;\; \mathit{if} \;\; C$$

because we can use the `owise` feature (described in Section 4.5.4) to cover all the remaining cases, when it is false, with an equation

$$x : \mathit{State} \models p(y_1, \ldots, y_n) = \mathit{false} \;\; \texttt{[owise]}.$$

In some cases, however—for example, because we want to perform reasoning using formal tools which do not accept `owise` declarations—we may fully define the true and false cases of a predicate not by using the `owise` attribute, but by explicit (possibly conditional) equations of the more general form,

$$t \models p(v_1, \ldots, v_n) = \mathit{bexp} \;\; \mathit{if} \;\; C,$$

where *bexp* is an arbitrary Boolean expression.

We are now ready to associate with a system module `M` specifying a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ (with a selected kind k of states and with state predicates Π defined by means of equations D in a protecting extension `M-PREDS`) a Kripke structure whose atomic predicates are specified by the set

$$AP_\Pi = \{\theta(p) \mid p \in \Pi,\ \theta \text{ ground substitution}\},$$

where by convention we use the simplified notation $\theta(p)$ to denote the ground term $\theta(p(x_1, \ldots, x_n))$. This defines a labeling function L_Π on the set of states $T_{\Sigma/E,k}$ assigning to each $[t] \in T_{\Sigma/E,k}$ the set of atomic propositions

$$L_\Pi([t]) = \{\theta(p) \in AP_\Pi \mid (E \cup D) \vdash \ (\forall\, \emptyset)\ t \models \theta(p) = \mathit{true}\}.$$

The Kripke structure we are interested in is then

$$\mathcal{K}(\mathcal{R},k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet, L_\Pi),$$

where $(\rightarrow^1_{\mathcal{R}})^\bullet$ denotes the total relation extending the one-step $\mathcal{R}$-rewriting relation $\rightarrow^1_{\mathcal{R}}$ among states of kind k, that is, $[t] \rightarrow^1_{\mathcal{R}} [t']$ holds if and only if there are $u \in [t]$ and $u' \in [t']$ such that u' is the result of applying one of the rules in R to u at some position. Under the usual assumptions that E is (ground) Church-Rosser and terminating (possibly modulo some axioms A contained in E) and R is (ground) coherent relative to E (again, possibly modulo A), u can always be chosen to be the canonical form of t under the equations E.

12.3 LTL model checking

Suppose that, given a system module `M` specifying a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, we have:

- chosen a kind k in `M` as our kind of states;
- defined some state predicates Π and their semantics in a module, say `M-PREDS`, protecting `M` by the method already explained in Section 12.2.

Then, as explained in Section 12.2, this defines a Kripke structure $\mathcal{K}(\mathcal{R},k)_\Pi$ on the set of atomic propositions AP_Π. Given an initial state $[t] \in T_{\Sigma/E,k}$ and an LTL formula $\varphi \in LTL(AP_\Pi)$ we would like to have a procedure to decide the satisfaction relation

$$\mathcal{K}(\mathcal{R},k)_\Pi, [t] \models \varphi.$$

In general this relation is undecidable. It becomes decidable if two conditions hold:

1. the set of states in $T_{\Sigma/E,k}$ that are *reachable* from $[t]$ by rewriting is *finite*,[1] and
2. the rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by `M` plus the equations D defining the predicates Π are such that:
   - both E and $E \cup D$ are (ground) Church-Rosser and terminating, perhaps modulo some axioms A, with $(\Sigma, E) \subseteq (\Sigma \cup \Pi, E \cup D)$ a protecting extension, and
   - R is (ground) coherent relative to E (again, perhaps modulo some axioms A).

Under these assumptions, both the state predicates Π and the transition relation $\rightarrow^1_{\mathcal{R}}$ are *computable* and, given the finite reachability assumption, we can then settle the above satisfaction problem using a *model-checking procedure*.

Specifically, Maude uses an on-the-fly LTL model-checking procedure of the style described in [24]. The basis of this procedure is the following. Each LTL formula φ has an associated Büchi automaton B_φ whose acceptance ω-language is exactly that of the behaviors satisfying φ. We can then reduce the satisfaction problem

$$\mathcal{K}(\mathcal{R},k)_\Pi, [t] \models \varphi$$

to the *emptiness problem* of the language accepted by the *synchronous product* of $B_{\neg\varphi}$ and (the Büchi automaton associated with) $(\mathcal{K}(\mathcal{R},k)_\Pi, [t])$. The formula φ is satisfied if and only if such a language is empty. The model-checking procedure checks emptiness by looking for a counterexample, that is, an infinite computation belonging to the language recognized by

[1] Note that this can happen even when $T_{\Sigma/E,k}$ is an infinite set: there is no requirement that $T_{\Sigma/E,k}$ is finite.

the synchronous product. For a detailed explanation of this general method of on-the-fly LTL model checking, see [24]; and for a discussion of the specific techniques used in the Maude LTL model-checker implementation, see [62].

This makes clear our interest in obtaining the *negative normal form* of a formula $\neg\varphi$, since we need it to build the Büchi automaton $B_{\neg\varphi}$. For efficiency purposes we need to make $B_{\neg\varphi}$ as small as possible. The following module `LTL-SIMPLIFIER` (also in the `model-checker.maude` file) tries to further simplify the negative normal form of the formula $\neg\varphi$ in the hope of generating a smaller Büchi automaton $B_{\neg\varphi}$. This module is optional (the user may choose to include it or not when doing model checking) but tends to help building a smaller $B_{\neg\varphi}$.

```
fmod LTL-SIMPLIFIER is
  including LTL .

  *** The simplifier is based on:
  ***  Kousha Etessami and Gerard J. Holzman,
  ***  "Optimizing Buchi Automata", CONCUR 2000, LNCS 1877.
  *** We use the Maude sort system to do much of the work.

  sorts TrueFormula FalseFormula PureFormula PE-Formula PU-Formula .
  subsort TrueFormula FalseFormula < PureFormula <
         PE-Formula PU-Formula < Formula .

  op True : -> TrueFormula [ctor ditto] .
  op False : -> FalseFormula [ctor ditto] .
  op _/\_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _/\_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _/\_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op _\/_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _\/_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _\/_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op O_ : PE-Formula -> PE-Formula [ctor ditto] .
  op O_ : PU-Formula -> PU-Formula [ctor ditto] .
  op O_ : PureFormula -> PureFormula [ctor ditto] .
  op _U_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _U_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _U_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op _U_ : TrueFormula Formula -> PE-Formula [ctor ditto] .
  op _U_ : TrueFormula PU-Formula -> PureFormula [ctor ditto] .
  op _R_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _R_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _R_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op _R_ : FalseFormula Formula -> PU-Formula [ctor ditto] .
  op _R_ : FalseFormula PE-Formula -> PureFormula [ctor ditto] .

  vars p q r s : Formula .
  var pe : PE-Formula .
  var pu : PU-Formula .
  var pr : PureFormula .

  *** Rules 1, 2 and 3; each with its dual.
  eq (p U r) /\ (q U r) = (p /\ q) U r .
  eq (p R r) \/ (q R r) = (p \/ q) R r .
  eq (p U q) \/ (p U r) = p U (q \/ r) .
  eq (p R q) /\ (p R r) = p R (q /\ r) .
```

```
eq True U (p U q) = True U q .
eq False R (p R q) = False R q .

*** Rules 4 and 5 do most of the work.
eq p U pe = pe .
eq p R pu = pu .

*** An extra rule in the same style.
eq O pr = pr .

*** We also use the rules from:
***  Fabio Somenzi and Roderick Bloem,
***  "Efficient Buchi Automata from LTL Formulae",
***  p247-263, CAV 2000, LNCS 1633.
*** that are not subsumed by the previous system.

*** Four pairs of duals.
eq O p /\ O q = O (p /\ q) .
eq O p \/ O q = O (p \/ q) .
eq O p U O q = O (p U q) .
eq O p R O q = O (p R q) .
eq True U O p = O (True U p) .
eq False R O p = O (False R p) .
eq (False R (True U p)) \/ (False R (True U q))
  = False R (True U (p \/ q)) .
eq (True U (False R p)) /\ (True U (False R q))
  = True U (False R (p /\ q)) .

*** <= relation on formula
op _<=_ : Formula Formula -> Bool [prec 75] .

eq p <= p = true .
eq False <= p = true .
eq p <= True = true .

ceq p <= (q /\ r) = true if (p <= q) /\ (p <= r) .
ceq p <= (q \/ r) = true if p <= q .
ceq (p /\ q) <= r = true if p <= r .
ceq (p \/ q) <= r = true if (p <= r) /\ (q <= r) .

ceq p <= (q U r) = true if p <= r .
ceq (p R q) <= r = true if q <= r .
ceq (p U q) <= r = true if (p <= r) /\ (q <= r) .
ceq p <= (q R r) = true if (p <= q) /\ (p <= r) .
ceq (p U q) <= (r U s) = true if (p <= r) /\ (q <= s) .
ceq (p R q) <= (r R s) = true if (p <= r) /\ (q <= s) .

*** conditional rules depending on <= relation
ceq p /\ q = p if p <= q .
ceq p \/ q = q if p <= q .
ceq p /\ q = False if p <= ~ q .
ceq p \/ q = True if ~ p <= q .
ceq p U q = q if p <= q .
ceq p R q = q if q <= p .
```

```
  ceq p U q = True U q if p =/= True /\ ~ q <= p .
  ceq p R q = False R q if p =/= False /\ q <= ~ p .
  ceq p U (q U r) = q U r if p <= q .
  ceq p R (q R r) = q R r if q <= p .
endfm
```

Suppose that all the requirements listed above to perform model checking are satisfied. How do we then model check a given LTL formula in Maude for a given initial state $[t]$ in a module `M`? We define a new module, say `M-CHECK`, according to the following pattern:

```
mod M-CHECK is
  protecting M-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER . *** optional
  op init : -> k .         *** optional
  eq init = t .            *** optional
endm
```

The declaration of a constant `init` of the kind of states is not necessary: it is a matter of convenience, since the initial state `t` may be a large term.

The module `MODEL-CHECKER` (in the file `model-checker.maude`) is as follows:

```
fmod MODEL-CHECKER is
  protecting QID .
  including SATISFACTION .
  including LTL .
  subsort Prop < Formula .

  *** transitions and results
  sorts RuleName Transition TransitionList ModelCheckResult .
  subsort Qid < RuleName .
  subsort Transition < TransitionList .
  subsort Bool < ModelCheckResult .
  ops unlabeled deadlock : -> RuleName .
  op {_,_} : State RuleName -> Transition [ctor] .
  op nil : -> TransitionList [ctor] .
  op __ : TransitionList TransitionList -> TransitionList
       [ctor assoc id: nil] .
  op counterexample :
       TransitionList TransitionList -> ModelCheckResult [ctor] .

  op modelCheck : State Formula ~> ModelCheckResult
       [special (...)] .
endfm
```

Its key operator is `modelCheck` (whose `special` attribute has been omitted here), which takes a state and an LTL formula and returns either the Boolean `true` if the formula is satisfied, or a counterexample when it is not satisfied. Note that the sort `Prop` coming from the `SATISFACTION` module is declared as a subsort of `Formula` in `LTL`. In each concrete use of `MODEL-CHECKER`, such as that in `M-CHECK` above, the atomic propositions in `Prop` will have been specified in a module such as `M-PREDS`.

Let us illustrate the use of this operator with our `MUTEX` example. Following the pattern described above, we can define the module

```
mod MUTEX-CHECK is
  protecting MUTEX-PREDS .
```

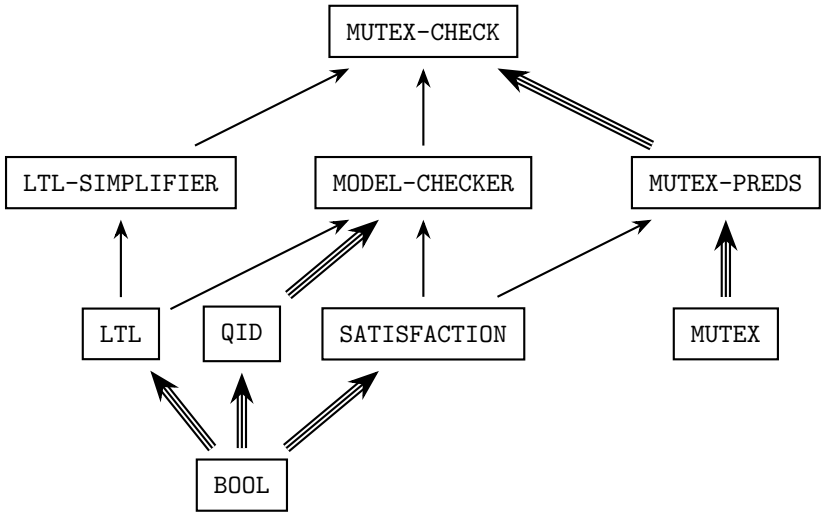


Figure 12.1: Importation graph of model-checking modules

```
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  ops initial1 initial2 : -> Conf .
  eq initial1 = $ [a, wait] [b, wait] .
  eq initial2 = * [a, wait] [b, wait] .
endm
```

The relationships between all the modules involved at this point is illustrated in Figure 12.1, where a single arrow represents an `including` importation and a triple arrow represents a `protecting` importation.

We are then ready to model check different LTL properties of `MUTEX`. The first obvious property to check is mutual exclusion:

```
Maude> red modelCheck(initial1, [] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK :
  modelCheck(initial1, []~ (crit(a) /\ crit(b))) .
result Bool: true

Maude> red modelCheck(initial2, [] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK :
  modelCheck(initial2, []~ (crit(a) /\ crit(b))) .
result Bool: true
```

We can also model check the strong liveness property that if a process waits infinitely often, then it is in its critical section infinitely often:

```
Maude> red modelCheck(initial1, ([]<> wait(a)) -> ([]<> crit(a))) .
reduce in MUTEX-CHECK :
  modelCheck(initial1, []<> wait(a) -> []<> crit(a)) .
result Bool: true

Maude> red modelCheck(initial1, ([]<> wait(b)) -> ([]<> crit(b))) .
reduce in MUTEX-CHECK :
```

```
  modelCheck(initial1, []<> wait(b) -> []<> crit(b)) .
result Bool: true

Maude> red modelCheck(initial2, ([]<> wait(a)) -> ([]<> crit(a))) .
reduce in MUTEX-CHECK :
  modelCheck(initial2, []<> wait(a) -> []<> crit(a)) .
result Bool: true

Maude> red modelCheck(initial2, ([]<> wait(b)) -> ([]<> crit(b))) .
reduce in MUTEX-CHECK :
  modelCheck(initial2, []<> wait(b) -> []<> crit(b)) .
result Bool: true
```

Of course, not all properties are true. Therefore, instead of a success we can get a *counterexample* showing why a property fails. Suppose that we want to check whether, beginning in the state `initial1`, process `b` will always be waiting. We then get the counterexample:

```
Maude> red modelCheck(initial1, [] wait(b)) .
reduce in MUTEX-CHECK : modelCheck(initial1, []wait(b)) .
result ModelCheckResult:
  counterexample({$ [a, wait] [b, wait], 'a-enter}
                 {[a, critical] [b, wait], 'a-exit}
                 {* [a, wait] [b, wait], 'b-enter},
                 {[a, wait] [b, critical], 'b-exit}
                 {$ [a, wait] [b, wait], 'a-enter}
                 {[a, critical] [b, wait], 'a-exit}
                 {* [a, wait] [b, wait], 'b-enter})
```

The main constructors used in the syntax of a counterexample term are:

```
op {_,_} : State RuleName -> Transition .
op nil : -> TransitionList [ctor] .
op __ : TransitionList TransitionList -> TransitionList
    [ctor assoc id: nil] .
op counterexample :
    TransitionList TransitionList -> ModelCheckResult [ctor] .
```

Therefore, a counterexample is a pair consisting of two lists of transitions, where the first corresponds to a finite path beginning in the initial state, and the second describes a loop. This is because, if an LTL formula φ is not satisfied by a finite Kripke structure, it is always possible to find a counterexample for φ having the form of a path of transitions followed by a cycle [24]. Note that each transition is represented as a *pair*, consisting of a state and the label of the rule applied to reach the next state.

Let us finish this section with an example of how *not to use* the model checker. Consider the following specification:

```
mod MODEL-CHECK-BAD-EX is
  including MODEL-CHECKER .
  extending LTL .
  sort Foo .
  op a : -> Foo .
  op f : Foo -> Foo .
  rl a => f(a) .

  subsort Foo < State .
  op p : -> Prop .
```

```
endm
```

This module describes an *infinite* transition system of the form

$$\texttt{a} \to \texttt{f(a)} \to \texttt{f(f(a))} \to \texttt{f(f(f(a)))} \to \cdots,$$

where the property `p` is not satisfied anywhere. Therefore the state `a` does not satisfy, e.g., the property `[]p`. However, if we try to invoke Maude with the command

```
Maude> red in MODEL-CHECK-BAD-EX : modelCheck(a, []p) .
```

we run into a nonterminating computation.

Making explicit that `p` does not hold in `a` by adding the equation

```
eq (a |= p) = false .
```

does not help. We run into the same problem even if the formula does not contain a temporal operator: `modelCheck(a, p)` does not terminate either. The reason is that the set of states reachable from `a` is *not* finite, and hence one of the main requirements for the model-checking algorithm is not satisfied.

12.4 The LTL satisfiability and tautology checker

A formula $\varphi \in \mathrm{LTL}(AP)$ is *satisfiable* if there is a Kripke structure $\mathcal{A} = (A, \to_{\mathcal{A}}, L)$, with $L : A \longrightarrow \mathcal{P}(AP)$, and a computation path π such that $\mathcal{A}, \pi \models \varphi$. Satisfiability of a formula $\varphi \in \mathrm{LTL}(AP)$ is a decidable property. In Maude, the satisfiability decision procedure is supported by the following predefined functional module `SAT-SOLVER` (also in the file `model-checker.maude`).

```
fmod SAT-SOLVER is
  protecting LTL .

  *** formula lists and results
  sorts FormulaList SatSolveResult TautCheckResult .
  subsort Formula < FormulaList .
  subsort Bool < SatSolveResult TautCheckResult .
  op nil : -> FormulaList [ctor] .
  op _;_ : FormulaList FormulaList -> FormulaList
      [ctor assoc id: nil] .
  op model : FormulaList FormulaList -> SatSolveResult [ctor] .

  op satSolve : Formula ~> SatSolveResult [special (...)] .

  op counterexample :
      FormulaList FormulaList -> TautCheckResult [ctor] .
  op tautCheck : Formula ~> TautCheckResult .
  op $invert : SatSolveResult -> TautCheckResult .

  var F : Formula .
  vars L C : FormulaList .
  eq tautCheck(F) = $invert(satSolve(~ F)) .
  eq $invert(false) = true .
  eq $invert(model(L, C)) = counterexample(L, C) .
endfm
```

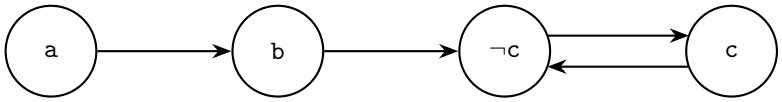


Figure 12.2: Graphical representation of a Kripke structure

One can define the desired atomic predicates in a module extending `SAT-SOLVER`, such as, for example,

```
fmod SAT-SOLVER-TEST is
  extending SAT-SOLVER .
  extending LTL .
  ops a b c d e p q r : -> Formula .
endfm
```

The user can then decide the satisfiability of an LTL formula involving those atomic propositions by applying the operator `satSolve` (whose `special` attribute has also been omitted in the module above) to the given formula and evaluating the expression. The resulting solution of sort `SatSolveResult` is then either `false`, if no model exists, or a finite model satisfying the formula. Such a model is described by a comma-separated pair of finite paths of states: an initial path leading to a cycle. Each state is described by a conjunction of atomic propositions or negated atomic propositions, with the propositions not mentioned in the conjunction being "don't care" ones. For example, we can evaluate

```
Maude> red satSolve(a /\ (O b) /\ (O O ((~ c) /\ [](c \/ (O c))))) .
reduce in SAT-SOLVER-TEST :
  satSolve(O O (~ c /\ [](c \/ O c)) /\ (a /\ O b)) .
result SatSolveResult: model(a ; b, (~ c) ; c)
```

which is satisfied by a four-state model with `a` holding in the first state, `b` holding in the second, `c` not holding in the third but holding in the fourth, and the fourth state going back to the third, as shown in Figure 12.2.

We call $\varphi \in \mathrm{LTL}(AP)$ a *tautology* if and only if $\mathcal{A}, a \models_{LTL} \varphi$ holds for every Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ with $L : A \longrightarrow \mathcal{P}(AP)$, and every state $a \in A$. It then follows easily that φ is a tautology if and only if $\neg\varphi$ is unsatisfiable. Therefore, the module `SAT-SOLVER` can also be used as a tautology checker. This is accomplished by using the `tautCheck`, `$invert`, and `counterexample` operators and their corresponding equations in `SAT-SOLVER`. The `tautCheck` function returns either `true` if the formula is a tautology, or a finite model that does not satisfy the formula. For example, we can evaluate:

```
Maude> red tautCheck((a => (O a)) <-> (a => ([] a))) .
reduce in SAT-SOLVER-TEST : tautCheck((a => O a) <-> a => []a) .
result Bool: true

Maude> red tautCheck((a -> (O a)) <-> (a -> ([] a))) .
reduce in SAT-SOLVER-TEST : tautCheck((a -> O a) <-> a -> []a) .
result TautCheckResult: counterexample(a ; a ; (~ a), True)
```

The tautology checker gives us also a *decision procedure for semantic* LTL *equality*, which is further discussed in [62].

12.5 Other model-checking examples

In [29, Section 16.6] some properties of a Mobile Maude application are model checked. This example is interesting because two levels of reflection (see Chapter 17) are involved: the object level, at which Mobile Maude system code executes, and the metalevel, at which application code is executed.

The model checker can also be executed in Full Maude. This is illustrated with an example in Section 22.7. This example, though quite simple, is interesting in several ways. The use of parameterization is exploited at both the system and the property level. At the system level, it allows a succinct specification of a parametric system. At the property level, it makes possible the specification of the relevant properties for each value of the parameter, also in a very succinct way. This is quite useful, because the property formulas vary as the parameter changes, and the parametric description encompasses an infinite number of instance formulas.

Chapter 13

Unification

13.1 Introduction

Unification is the solving of equations, either in *free algebras* of the form $T_\Sigma(X)$, or in *relatively free algebras* modulo a set E of equations, that is, in algebras of the form $T_{\Sigma/E}(X)$. The first case is sometimes called *syntactic unification*. The second case is sometimes called *E-unification* or unification *modulo* E; if E is not explicitly mentioned, then it is called *equational unification* or *semantic unification*.

In solving any equation, such as, for example,

$$f(x, h(y)) = f(g(y), z)$$

we look for *instances* of the variables appearing in the equation that make both sides equal. Variables can of course be instantiated by *substitutions*. A substitution that makes both sides of the equation equal, that is, a solution of the equation, is called a *unifier*. For example, if we are solving the above equation in the free algebra $T_\Sigma(X)$ with X a countable set of variables and with Σ having a single sort (unsorted unification), the substitution $\sigma = \{x \mapsto g(y), z \mapsto h(y)\}$ is a unifier, and indeed the so-called *most general unifier*, so that for any other unifier ρ there exists a substitution μ such that $\rho = \sigma; \mu$, where $\sigma; \mu$ denotes composition of substitutions in diagrammatic order. That is, any other solution of the equation is an *instance* of the most general solution provided by σ.

Of course, some equations may not have syntactic unifiers, but may have semantic unifiers *modulo* some equations E. Consider, for example, the equation

$$f(h(y), x) = f(g(y), z)$$

which obviously does not have any solution in $T_\Sigma(X)$. It does, however, have a solution in $T_{\Sigma/C}(X)$, where C is the commutativity axiom $f(x, y) = f(y, x)$. Indeed, the exact same substitution σ solving the first equation $f(x, h(y)) = f(g(y), z)$ in a syntactic way, is now a unifier solving the second equation $f(h(y), x) = f(g(y), z)$ *modulo* C, because we have $f(h(y), g(y)) =_C f(g(y), h(y))$.

Unification is a fundamental deductive mechanism used in many automated deduction tasks (see Section 13.5 for a discussion of some of them). It is also very important in combining the paradigms of functional programming and logic programming (in the Prolog sense of "logic programming"). Furthermore, in the context of Maude, unification can be very useful to reason not only about equational theories (functional modules or theories), but also, as explained in Section 13.5.2, about rewrite theories (system modules or theories).

Therefore, it is very useful to have an efficient implementation of unification available in Core Maude, which is what this chapter describes. Specifically, we explain how order-sorted unification modulo frequently occurring equational axioms is supported in Maude. In Chapter 14, we explain order-sorted unification modulo convergent equational theories, also available in Maude.

13.2 Order-sorted unification

Although the most general equational theories supported by Maude are membership equational theories, to obtain *practical unification algorithms*, allowing us to effectively compute the solutions of an equational unification problem, it is important to restrict ourselves to *order-sorted* equational theories. We can define the basic concepts of order-sorted E-unification in full generality.

Given an order-sorted equational theory (Σ, E), an *E-unification problem* consists of a nonempty set of unification equations of the form $t =^? t'$, written in the notation

$$t_1 =^? t'_1 \wedge \ldots \wedge t_n =^? t'_n$$

where $n \geq 1$ and the "conjunction" operator $\wedge$ is assumed to be associative and commutative.

Given such an E-unification problem, an *E-unifier* for it is an order-sorted substitution[1] $\sigma : \mathit{Vars}(t_1, t'_1, \ldots, t_n, t'_n) \longrightarrow T_\Sigma(X)$ (where we assume that the set X of variables contains a countable number of variables for each sort) such that, for all $i = 1, \ldots, n$,

$$E \vdash (\forall Y_i)\ \sigma(t_i) = \sigma(t'_i),$$

where $Y_i = \mathit{Vars}(\sigma(t_i), \sigma(t'_i))$, that is, all the equations $(\forall Y_i)\ \sigma(t_i) = \sigma(t'_i)$ can be deduced in (membership) equational logic from the set of equations E.

A set $\mathcal{U}$ of unifiers is called a *complete set of E-unifiers* for a given E-unification problem $t_1 =^? t'_1 \wedge \ldots \wedge t_n =^? t'_n$ iff for any other E-unifier ρ of the same E-unification problem there exists a substitution μ and a unifier $\sigma \in \mathcal{U}$ such that $\rho =_E \sigma; \mu$, that is, for each variable x in the domain of ρ we have $E \vdash \rho(x) = \mu(\sigma(x))$. A complete set $\mathcal{U}$ of E-unifiers is called *minimal* if any proper subset of $\mathcal{U}$ fails to be complete.

For an order-sorted equational theory (Σ, E), unification is said to be *finitary* if for any E-unification problem there is always a finite complete set of unifiers $\mathcal{U}$. Similarly, unification for (Σ, E) is called *unitary* if it is finitary and for any E-unification problem a minimal complete set of unifiers is always either empty or a singleton set. We say that (Σ, E) has a *unification algorithm* if there is an algorithm generating a complete set of E-unifiers for any E-unification problem in (Σ, E).

Unlike unsorted syntactic unification, which always either fails or has a single most general unifier, order-sorted syntactic unification is *not* necessarily unitary, that is, there is in general no single most general unifier. What exists (if Σ is finite) is a finite minimal complete set of syntactic unifiers. For some commonly occurring theories having a unification algorithm, such as the theory A of associativity of a binary function symbol, it is well-known that unification is not finitary: in general an infinite number of solutions may exist. However, for other theories, such as commutativity C or associativity-commutativity AC, unification is finitary, both when Σ is unsorted and when Σ is order-sorted (and finite).

[1] That is, a sort-preserving mapping from $\mathit{Vars}(t_1, t'_1, \ldots, t_n, t'_n)$, the set of all variables appearing in the terms t_i or t'_i, to $T_\Sigma(X)$.

13.2.1 A hybrid approach to equational order-sorted unification

We define a *hybrid* approach, in which we take advantage of Maude's structuring of a module's equations into equational axioms Ax, such as associativity, and/or commutativity, and/or identity, and equations E, which are assumed to be coherent,[2] confluent, and terminating modulo Ax. We can then consider order-sorted equational theories of the form $(\Sigma, E \cup Ax)$ and *decompose* the $E \cup Ax$-unification problem into *two* problems: one of Ax-unification, and another of $E \cup Ax$-unification that uses an Ax-unification algorithm as a subroutine. This decomposition, as well as a similar one for membership equational theories, is explained in Section 13.5. The point of this decomposition is that Ax-unification needs to be built-in at the level of Core Maude's C++ implementation for efficiency purposes and $E \cup Ax$-unification can then be built on top of Ax-unification. Since the axioms Ax are well-known and unification algorithms exist for them, the task of building in efficient Ax-unification algorithms, although highly nontrivial, becomes manageable. Maude 2.6 implemented $E \cup Ax$-unification in Maude itself, but $E \cup Ax$-unification is also implemented in Core Maude's C++ level since Maude 2.7 (see Chapter 14).

13.3 Theories currently supported

As mentioned in Section 13.2.1, a practical way of dealing with order-sorted equational unification is to consider order-sorted theories of the form $(\Sigma, E \cup Ax)$, with Ax a set of commonly occurring axioms, declared in Maude as equational attributes (see Section 4.4.1), and E the remaining equations specified with the `eq` or `ceq` keywords. We can then decompose the $E \cup Ax$-unification problem into an Ax-unification problem and an $E \cup Ax$-unification problem that uses an Ax-unification algorithm as a subroutine. In such a decomposition, the efficiency of the Ax-unification algorithm becomes crucial.

Maude currently provides an order-sorted Ax-unification algorithm for all order-sorted theories $(\Sigma, E \cup Ax)$ such that the order-sorted signature Σ is *preregular* modulo Ax (see Sections 3.8 and 20.3.5) and the axioms Ax associated to function symbols are as follows:

- there can be arbitrary function symbols and constants with no equational attributes;
- the `iter` equational attribute can be declared for some unary symbols;
- different equational attributes can be declared for some binary function symbols:
  - the `assoc` attribute[3]; usually referred to as A (for associative),
  - the `comm` attribute; usually referred to as C (for commutative),
  - the `assoc comm` attributes; usually referred to as AC (for associative and commutative),

[2] Coherence of E modulo Ax is very closely related to the notion of coherence of rules relative to equations explained in Section 5.3; see [85, 107] for the precise definition of coherence in the equational case. The main role of coherence of E modulo Ax is to get the effect of rewriting in Ax-equivalence classes with E. For example, for $Ax = AC$, coherence modulo AC is easily achieved by adding to each equation in E with a top AC function symbol in its lefthand side a similar equation with an extra "extension variable" argument added to the AC function symbol, as explained in Section 4.8. Section 4.8 also explains how, for rewriting modulo axioms Ax supported by Maude, Maude automatically performs such a coherence completion in an implicit, built-in way using extension variables and "extension aware" Ax-matching algorithms; see also Section 21.5.

[3] Although associative unification is infinitary in general, we support a substantial class of equational problems for which it is finitary, and provide a finite but incomplete set of solutions (with an explicit warning that it may be incomplete) in all other cases. See Section 13.4.6.

- the `assoc comm id:` attributes; usually referred to as ACU (for associative and commutative with identity, or unit),
- the `assoc id:` attributes[3]; usually referred to as AU (for associative and identity, or unit),
- the `comm id:` attributes; usually referred to as CU (for commutative and identity, or unit),
- the `id:` attribute; usually referred to as U (for identity, or unit),
- the `left id:` attribute; usually referred to as Ul (for left identity, or unit), and
- the `right id:` attribute; usually referred to as Ur (for right identity, or unit),

but no other equational attributes must be given for such symbols.

Explicitly excluded are theories with a binary symbol declared with the `idem` attribute. However, the lack of the `idem` attribute in general can be avoided by observing that this axiom has the finite variant property, so that, used as variant equations, it can be combined with all the other just-mentioned attributes by means of variant unification algorithms (see Section 14.8).

If we give to Maude a unification problem in a functional module of the form `fmod` (Σ, Ax) `endfm` where Σ and Ax satisfy the above requirements, we get a complete[4] set of order-sorted unifiers modulo the theory (Σ, Ax). If, instead, we give the same problem to Maude in the functional module `fmod` $(\Sigma, E \cup Ax)$ `endfm`, then the equations E are ignored, and we get the same unifiers as for the module or theory obtained by omitting the equations E. Similarly, if we provide the same unification problem in a functional theory `fth` $(\Sigma, E \cup Ax)$ `endfth`, a system module `mod` $(\Sigma, E \cup Ax, R)$ `endm` or a system theory `th` $(\Sigma, E \cup Ax, R)$ `endth`, we again get the same set of unifiers as for the theory (Σ, Ax). All this is consistent with the decomposition idea mentioned above: to deal with order-sorted $E \cup Ax$-unification, other methods, that use the Ax-unification algorithm as a component, can later be defined as we explain in Section 13.5.

Maude is even more tolerant than this. The user can give to Maude a unification problem in a functional module (or functional theory, or system module, or system theory) of the form `fmod` $(\Sigma, E \cup M \cup Ax \cup Ax')$ `endfm` (or the analogous specification in the other cases), where (Σ, Ax) satisfies the conditions mentioned above, but M is an optional set of *membership axioms* (that is, $(\Sigma, E \cup M \cup Ax \cup Ax')$ can be a membership equational theory and not just an order-sorted theory), and the axioms Ax' violate those conditions mentioned above. Then what will happen is:

1. As before, the additional equations E (or rules R) are completely ignored, and the membership axioms M are likewise ignored.

2. If a unification problem involves the occurrence of a symbol satisfying axioms Ax' at the root position of a *non-ground subterm*, the unification process will *fail* and a warning will be printed.

3. If a unification problem involves the occurrence of symbols satisfying axioms Ax', but all such occurrences are always in *ground* subterms of the problem, then this special case of $Ax \cup Ax'$-unification is handled by Maude and the corresponding $Ax \cup Ax'$-unifiers are returned.

[4] Or a finite but incomplete set (with an explicit warning) when one of the symbols has the `assoc` attribute without the `comm` attribute and the equations fall outside the class for which a finite, complete set of unifiers can be computed.

Furthermore, the functional module `fmod` $(\Sigma, E \cup M \cup Ax \cup Ax')$ `endfm` (or the analogous functional theory or system module or theory) may import predefined modules such as `BOOL` or `NAT`, so that function symbols in such predefined modules can also be used in unification problems.

13.4 The `unify` command

Given a functional module or theory, or a system module or theory, ⟨*ModId*⟩, the user can give to Maude a unification command of the following two forms:

```
unify [ n ] in ⟨ModId⟩ :
  ⟨Term-1⟩ =? ⟨Term'-1⟩ /\ ... /\ ⟨Term-k⟩ =? ⟨Term'-k⟩ .
irredundant unify [ n ] in ⟨ModId⟩ :
  ⟨Term-1⟩ =? ⟨Term'-1⟩ /\ ... /\ ⟨Term-k⟩ =? ⟨Term'-k⟩ .
```

where $k \geq 1$; n is an optional argument providing a bound on the number of unifiers requested, so that if the cardinality of the set of unifiers is greater than the specified bound, the unifiers beyond that bound are omitted; and ⟨*ModId*⟩ can be any module or theory declared in the current Maude session (as usual, if no module is mentioned, the current module is used). The second command generates all the unifiers and, then, filters them against each other in order to return a minimal set of most general unifiers modulo the axioms.

For a simple example of syntactic order-sorted unification problem illustrating:

- the use of the `unify` command;
- the use of the predefined operator `_^_` in the `NAT` module, representing exponentiation on natural numbers; and
- the, in general, non-unitary nature of order-sorted unification,

we can define the module

```
fmod UNIFICATION-EX1 is
  protecting NAT .
  op f : Nat Nat -> Nat .
  op f : NzNat Nat -> NzNat .
  op f : Nat NzNat -> NzNat .
endfm
```

and then give to Maude the following command:

```
Maude> unify f(X:Nat, Y:Nat) ^ B:NzNat =? A:NzNat ^ f(Y:Nat, Z:Nat) .

Unifier 1
X:Nat --> #1:Nat
Y:Nat --> #2:NzNat
B:NzNat --> f(#2:NzNat, #3:Nat)
A:NzNat --> f(#1:Nat, #2:NzNat)
Z:Nat --> #3:Nat

Unifier 2
X:Nat --> #1:NzNat
Y:Nat --> #2:Nat
B:NzNat --> f(#2:Nat, #3:NzNat)
A:NzNat --> f(#1:NzNat, #2:Nat)
Z:Nat --> #3:NzNat
```

The next example in the same module illustrates the use of the `unify` command with a unification problem consisting of two equations:

```
Maude> unify f(X:Nat, Y:NzNat) =? f(Z:NzNat, U:Nat)
        /\ V:NzNat =? f(X:Nat, U:Nat) .

Unifier 1
X:Nat --> #1:NzNat
Y:NzNat --> #2:NzNat
Z:NzNat --> #1:NzNat
U:Nat --> #2:NzNat
V:NzNat --> f(#1:NzNat, #2:NzNat)
```

Note that, as already mentioned, we could instead invoke the `unify` command in a functional or system module or theory having additional equations, memberships, or rules, which are always ignored. For example, we could have instead declared the system theory

```
th UNIFICATION-EX2 is
  protecting NAT .
  op f : Nat Nat -> Nat .
  op f : NzNat Nat -> NzNat .
  op f : Nat NzNat -> NzNat .
  eq f(f(N:Nat, M:Nat), K:Nat) = f(N:Nat, M:Nat) .
  rl f(N:Nat, M:Nat) => 0 .
endth
```

so that, if we give again the same `unify` commands above, we will obtain *exactly* the same sets of order-sorted unifiers as for the `UNIFICATION-EX1` module.

The following unification command in the predefined `CONVERSION` module (see Section 7.9) illustrates a further point on the handling of built-in constants and functions. Built-in constants, even though infinite in number (all strings, all quoted identifiers, all natural numbers, and so on), are handled and can be used in unification problems. But built-in functions are not considered built-in for unification purposes; therefore, built-in evaluation of such functions is *not* performed during the unification.

```
Maude> unify in CONVERSION :
        X:String < "foo" + Y:Char =?
          Z:String + string(pi) < "foo" + Z:String .

Unifier 1
X:String --> #1:Char + string(pi)
Y:Char --> #1:Char
Z:String --> #1:Char
```

The above examples illustrate a further point about the form of the returned unifiers, namely, that in each assignment X `-->` t in a unifier, the variables appearing in the term t are always *fresh* variables of the form `#n:Sort`. The user is required[5] *not to use* variables of this form in the submitted unification problem. A warning is printed if this requirement is violated:

```
Maude> unify in NAT : X:Nat ^ #1:Nat =? #2:Nat .
Warning: unsafe variable name #1:Nat in unification problem.
```

[5]When unification is used at the metalevel (see Section 17.6.8), there are two possibilities: (i) a counter for new fresh variables must take into account the numbers used for both forms of fresh variables, or (ii) an alternative identifier for new variables can be given.

13.4.1 Non-supported unification examples

The handling of unification problems in modules with operators whose equational axioms are excluded from the current unification algorithm can be illustrated by means of the following module:

```
fmod UNIFICATION-EX3 is
  protecting NAT .
  op f : Nat Nat -> Nat [idem] .
endfm
```

As already mentioned, a unification problem using such an idempotent function symbol `f` in a non-ground subterm will mean that the unification attempt fails and a warning is issued:

```
Maude> unify f(f(X:Nat, Y:Nat), Z:Nat) =? f(A:Nat, B:Nat) .
Warning: Term f(X:Nat, Y:Nat, Z:Nat) is non-ground and unification
  for its top symbol is not currently supported.
```

Instead, if all symbols satisfying unsupported equational axioms Ax' only appear in *ground* subterms of the unification problem, the unification attempt succeeds with the correct set of order-sorted $Ax \cup Ax'$-unifiers:

```
Maude> unify X:Nat + f(f(41, 42),43) =? Y:Nat + f(41,f(42,43)) .

Unifier 1
X:Nat --> #1:Nat + f(41, f(42, 43))
Y:Nat --> #1:Nat + f(f(41, 42), 43)

Unifier 2
X:Nat --> f(41, f(42, 43))
Y:Nat --> f(f(41, 42), 43)
```

Note, however, that, as already mentioned, unification modulo the `idem` attribute can be achieved by variant-based unification using the explicit equation `f(N:Nat,N:Nat) = N:Nat` (see Section 14.8).

13.4.2 Associative-commutative (AC) unification examples

The use of a bound on the number of unifiers, as well as the use of the associative-commutative (AC) operator `+` in the predefined `NAT` module (see Section 7.2), plus the fact that even small AC-unification problems can generate a large number of unifiers are all illustrated by the following command:

```
Maude> unify [100] in NAT :
        X:Nat + X:Nat + Y:Nat =? A:Nat + B:Nat + C:Nat .
Unifier 1
X:Nat --> #1:Nat + #2:Nat + #3:Nat + #5:Nat + #6:Nat + #8:Nat
Y:Nat --> #4:Nat + #7:Nat + #9:Nat
A:Nat --> #1:Nat + #1:Nat + #2:Nat + #3:Nat + #4:Nat
B:Nat --> #2:Nat + #5:Nat + #5:Nat + #6:Nat + #7:Nat
C:Nat --> #3:Nat + #6:Nat + #8:Nat + #8:Nat + #9:Nat
...

Unifier 100
X:Nat --> #1:Nat + #2:Nat + #3:Nat + #4:Nat
Y:Nat --> #5:Nat
```

```
A:Nat --> #1:Nat + #1:Nat + #2:Nat
B:Nat --> #2:Nat + #3:Nat
C:Nat --> #3:Nat + #4:Nat + #4:Nat + #5:Nat
```

The unification problem above has 381 unifiers, though only 100 were asked for. Indeed, it is the minimal set of most general unifiers and the `irredundant unify` command returns the same set with 381 unifiers.

13.4.3 Unification examples with the `iter` attribute

The following example illustrates the efficiency of order-sorted unification modulo the `iter` theory (in this example in combination with the `comm` theory). Consider the following module:

```
fmod ITER-EXAMPLE is
  sorts NzEvenNat EvenNat OddNat NzNat Nat EvenInt OddInt NzInt Int .
  subsorts OddNat < OddInt NzNat < NzInt < Int .
  subsorts EvenNat < EvenInt Nat < Int .
  subsorts NzEvenNat < NzNat EvenNat < Nat .

  op 0 : -> EvenNat .

  op s : EvenNat -> OddNat [iter] .
  op s : OddNat -> NzEvenNat [iter] .
  op s : Nat -> NzNat [iter] .
  op s : EvenInt -> OddInt [iter] .
  op s : OddInt -> EvenInt [iter] .
  op s : Int -> Int [iter] .

  op _+_ : Int Int -> Int [comm gather (E e)] .
  op _+_ : OddInt OddInt -> EvenInt [ditto] .
  op _+_ : EvenInt EvenInt -> EvenInt [ditto] .
  op _+_ : OddInt EvenInt -> OddInt [ditto] .
  op _+_ : Nat Nat -> Nat [ditto] .
  op _+_ : Nat NzNat -> NzNat [ditto] .
  op _+_ : OddNat OddNat -> NzEvenNat [ditto] .
  op _+_ : NzEvenNat EvenNat -> NzEvenNat [ditto] .
  op _+_ : EvenNat EvenNat -> EvenNat [ditto] .
  op _+_ : OddNat EvenNat -> OddNat [ditto] .
endfm
```

We can then give the unification commands:

```
Maude> unify in ITER-EXAMPLE :
        s^1000000(X:OddNat) =? s^100000000001(Y:Int) .
Decision time: 1ms cpu (1ms real)

Unifier 1
X:OddNat --> s^99999000001(#1:EvenNat)
Y:Int --> #1:EvenNat
```

and

```
Maude> unify in ITER-EXAMPLE :
        s^1000000(X:OddNat) =? s^100000000001(Y:Int + Z:Int + W:Int) .
Decision time: 2ms cpu (5ms real)
```

```
Unifier 1
X:OddNat --> s^99999000001(#1:OddNat + (#2:OddNat + #3:EvenNat))
W:Int --> #1:OddNat
Z:Int --> #2:OddNat
Y:Int --> #3:EvenNat

Unifier 2
X:OddNat --> s^99999000001(#1:OddNat + (#2:EvenNat + #3:OddNat))
W:Int --> #1:OddNat
Z:Int --> #2:EvenNat
Y:Int --> #3:OddNat

Unifier 3
X:OddNat --> s^99999000001(#1:EvenNat + (#2:OddNat + #3:OddNat))
W:Int --> #1:EvenNat
Z:Int --> #2:OddNat
Y:Int --> #3:OddNat

Unifier 4
X:OddNat --> s^99999000001(#1:EvenNat + (#2:EvenNat + #3:EvenNat))
W:Int --> #1:EvenNat
Z:Int --> #2:EvenNat
Y:Int --> #3:EvenNat
```

Note that the second command produces more unifiers than the first command even though both unification problems have only one most general unifier. We can, however, obtain just the single most general unifier for the second unification problem by giving the command:

```
Maude> irredundant unify in ITER-EXAMPLE :
         s^1000000(X:OddNat) =? s^100000000001(Y:Int + Z:Int + W:Int) .
Decision time: 0ms cpu (0ms real)

Unifier 1
X:OddNat --> s^99999000001(#1:EvenNat + (#2:EvenNat + #3:EvenNat))
W:Int --> #1:EvenNat
Z:Int --> #2:EvenNat
Y:Int --> #3:EvenNat
```

As already mentioned, assuming that no bound on the number of unifiers is specified by the user, Maude will always compute a *complete* set of order-sorted unifiers modulo Ax, for Ax the supported equational axioms described in Section 13.3 for which unification is known to be finitary. However, there is no guarantee that the computed set of unifiers is *minimal* if the `unify` command, instead of the `irredundant unify` command, is used. That is, some of the unifiers in the computed set may be redundant, since they could be obtained as instances (modulo Ax) of other unifiers in the set.

13.4.4 Associative-commutative with identity (ACU) unification examples

To illustrate the use of the unification command in the presence of ACU operators, let us consider yet another version of the vending machine example (first presented in Section 5.1 and in other sections of this document in different forms):

```
mod UNIF-VENDING-MACHINE is
```

```
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : < M $ > => < M c > .
  rl [buy-a] : < M $ > => < M a q > .
  eq [change]: q q q q = $ .
endm
```

We can ask whether there is an equational unifier of two configurations, one containing at least two quarters, and another containing at least one dollar.

```
Maude> unify in UNIF-VENDING-MACHINE :
        < q q X:Marking > =? < $ Y:Marking > .

Unifier 1
X:Marking --> $
Y:Marking --> q q

Unifier 2
X:Marking --> $ #1:Marking
Y:Marking --> q q #1:Marking
```

Notice that the computed set of unifiers is not minimal, because the first solution is the instance of the second obtained by substituting the variable `#1:Marking` with the constant `empty`.

```
Maude> irredundant unify in UNIF-VENDING-MACHINE :
        < q q X:Marking > =? < $ Y:Marking > .

Unifier 1
X:Marking --> $ #1:Marking
Y:Marking --> q q #1:Marking
```

Recall that memberships are discarded completely. For instance, if we modify the previous example to include a membership definition for a new sort `Quarter`, any unification call with that sort may not succeed.

```
mod UNIF-VENDING-MACHINE-MB is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .

  sort Quarter .
  subsort Quarter < Coin .
```

```
  mb q : Quarter .

  var M : Marking .
  rl [buy-c] : < M $ > => < M c > .
  rl [buy-a] : < M $ > => < M a q > .
  eq [change]: q q q q = $ .
endm
```

We can ask whether there is an equational unifier of two configurations, one containing at least two quarters, and another containing two quarters and a dollar, but it fails:

```
Maude> unify in UNIF-VENDING-MACHINE-MB :
        < q q X:Marking > =? < $ Y:Quarter Z:Quarter > .

No unifier.
```

despite the fact that instantiating both `Y` and `Z` to `q` is part of a solution in the unification call above. The reason is that the membership is not used during ACU unification and therefore the algorithm unification treats the sort `Quarter` as empty.

13.4.5 Unification examples with an identity symbol

Let us illustrate the use of the different combinations of the identity attribute for unification. Let us consider first a module using the `left-id` attribute.

```
mod LEFTID-UNIFICATION-EX is
    sorts Magma Elem .
    subsorts Elem < Magma .
    op __ : Magma Magma -> Magma [gather (E e) left id: e] .
    ops a b c d e : -> Elem .
endm
```

Then the following two unification problems have a different meaning, where we have swapped the position of the variables. First, when we unify two terms where variables of sort `Magma` are at the left of the terms, we have both a syntactic unifier and a unifier modulo identity.

```
Maude> unify in LEFTID-UNIFICATION-EX : X:Magma a =? Y:Magma a a .

Unifier 1
X:Magma --> a
Y:Magma --> e

Unifier 2
X:Magma --> #1:Magma a
Y:Magma --> #1:Magma
```

When the variables are instead at the right side of the terms of sort `Magma`, there is clearly no unifier, since the term `a a Y:Magma` is parsed as `(a a) Y:Magma` in module `LEFTID-UNIFICATION-EX` due to the attribute `gather (E e)` (see Section 3.9).

```
Maude> unify in LEFTID-UNIFICATION-EX : a X:Magma =? a a Y:Magma .
No unifier.
```

Consider now a similar module but for the right identity.

```
mod RIGHTID-UNIFICATION-EX is
    sorts Magma Elem .
```

```
    subsorts Elem < Magma .
    op __ : Magma Magma -> Magma [gather (e E) right id: e] .
    ops a b c d e : -> Elem .
endm
```

When we unify two terms where variables of sort `Magma` are at the left of the terms, there is clearly no unifier, since the term `Y:Magma a a` is parsed this time as `Y:Magma (a a)` in module `RIGHTID-UNIFICATION-EX` due to the attribute `gather (e E)` (see Section 3.9).

```
Maude> unify in RIGTHID-UNIFICATION-EX : X:Magma a =? Y:Magma a a .
No unifier.
```

When the variables are instead at the right side of the terms of sort `Magma`, we have both a syntactic unifier and a unifier modulo identity:

```
Maude> unify in RIGTHID-UNIFICATION-EX : a X:Magma =? a a Y:Magma .

Unifier 1
X:Magma --> a
Y:Magma --> e

Unifier 2
X:Magma --> #1:Magma a
Y:Magma --> #1:Magma
```

Consider now a similar module but with the identity attribute.

```
mod ID-UNIFICATION-EX is
    sorts Magma Elem .
    subsorts Elem < Magma .
    op __ : Magma Magma -> Magma [gather (E e) id: e] .
    ops a b c d e : -> Elem .
endm
```

When we unify two terms where variables of sort `Magma` are at the left of the terms, we have both a syntactic unifier and a unifier modulo identity:

```
Maude> unify in ID-UNIFICATION-EX : X:Magma a =? Y:Magma a a .

Unifier 1
X:Magma --> a
Y:Magma --> e

Unifier 2
X:Magma --> #1:Magma a
Y:Magma --> #1:Magma
```

When the variables of sort `Magma` are instead at the right side of the terms of sort `Magma`, we only have a unifier modulo identity:

```
Maude> unify in ID-UNIFICATION-EX : a X:Magma =? a a Y:Magma .

Unifier 1
X:Magma --> a
Y:Magma --> e
```

And finally, when we add commutativity, we obtain slightly different results.

```
mod COMM-ID-UNIFICATION-EX is
    sorts Magma Elem .
    subsorts Elem < Magma .
    op __ : Magma Magma -> Magma [gather (E e) comm id: e] .
    ops a b c d e : -> Elem .
endm
```

When we unify two terms where variables of sort `Magma` are at the left of the terms, we have both a syntactic unifier and a unifier modulo identity and commutativity, but the latter is duplicated:

```
Maude> unify in COMM-ID-UNIFICATION-EX : X:Magma a =? Y:Magma a a .

Unifier 1
X:Magma --> a
Y:Magma --> e

Unifier 2
X:Magma --> a #1:Magma
Y:Magma --> #1:Magma

Unifier 3
X:Magma --> a
Y:Magma --> e
```

When the variables of sort `Magma` are instead at the right side of the terms of sort `Magma`, we have several unifiers modulo identity and commutativity:

```
Maude> unify in COMMID-UNIFICATION-EX : a X:Magma =? a a Y:Magma .

Unifier 1
X:Magma --> a a
Y:Magma --> a

Unifier 2
X:Magma --> a
Y:Magma --> e

Unifier 3
X:Magma --> a
Y:Magma --> e
```

Note that the first solution is intriguing and is obtained by unifying terms `(X:Magma a)` and `((a a) Y:Magma)`.

13.4.6 Associative (A) unification examples

In general, unification modulo associativity is not finitary. However the Maude implementation provides a finitary and complete set of unifiers for a substantial class of unification problems where associative unification happens to be finitary. In problems outside of this class, a finite but in general incomplete set of unifiers is returned, with an explicit warning that the set may be incomplete. Let us illustrate the use of the unification command in the presence of A symbols with the different capabilities and limitations. One of the key ideas for having finitary unification modulo associativity is to have variables over associative symbols being linear, i.e., having only one occurrence among all the terms in a unification problem.

Consider a very simple module with a binary associative function symbol:

```
fmod UNIFICATION-EX4 is
  protecting NAT .
  sort NList .
  subsort Nat < NList .
  op _:_ : NList NList -> NList [assoc] .
endfm
```

A unification problem using such an associative function symbol returns five unifiers without any problem because the unification problem is linear.

```
Maude> unify in UNIFICATION-EX4 : X:NList : Y:NList : Z:NList =? P:NList : Q:NList .

Unifier 1
X:NList --> #1:NList : #2:NList
Y:NList --> #3:NList
Z:NList --> #4:NList
P:NList --> #1:NList
Q:NList --> #2:NList : #3:NList : #4:NList

Unifier 2
X:NList --> #1:NList
Y:NList --> #2:NList : #3:NList
Z:NList --> #4:NList
P:NList --> #1:NList : #2:NList
Q:NList --> #3:NList : #4:NList

Unifier 3
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList : #4:NList
P:NList --> #1:NList : #2:NList : #3:NList
Q:NList --> #4:NList

Unifier 4
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList
P:NList --> #1:NList : #2:NList
Q:NList --> #3:NList

Unifier 5
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList
P:NList --> #1:NList
Q:NList --> #2:NList : #3:NList
```

The unification problem may not be linear but it may be easy to detect that there is no unifier, e.g. it is impossible to unify a list `X` concatenated with itself with another list `Y` concatenated also with itself but with a natural number, e.g. 1, in between.

```
Maude> unify in UNIFICATION-EX4 : X:NList : X:NList =? Y:NList : 1 : Y:NList .
No unifier.
```

When associative variables are non-linear, Maude has implemented a cycle detection that may have different outcomes:

1. There are non-linear variables but only on one side of the problem. This implies that there is no risk of infinitary behavior and Maude will be complete with no need for cycle detection.

```
Maude> unify in UNIFICATION-EX4 : P:NList : P:NList =? 1 : Q:NList : 2 .

Unifier 1
P:NList --> 1 : #1:NList : 2
Q:NList --> #1:NList : 2 : 1 : #1:NList

Unifier 2
P:NList --> 1 : 2
Q:NList --> 2 : 1
```

2. Cycle detection is used when non-linear variables appear on both sides but no variable occurs more than twice. It may detect spurious cycles, i.e., a cycle that does not correspond to unifiers, but then nothing is reported. If there is at least one unifier associated to the cycle, a warning will be printed and the acyclic solutions are returned.

```
Maude> unify in UNIFICATION-EX4 : 0 : X:NList =? X:NList : 0 .
Warning: Unification modulo the theory of operator _:_ has encountered
an instance for which it may not be complete.

Unifier 1
X:NList --> 0
Warning: Some unifiers may have been missed due to incomplete
unification algorithm(s).
```

   Note that the unification problem $\mathtt{0 : X} =^{?} \mathtt{X : 0}$ has an infinite family of most general unifiers $\{\mathtt{X} \mapsto \mathtt{0}^n\}$ for $\mathtt{0}^n$ being a list of n consecutive `0` elements.

3. If neither of the two above cases apply, then Maude forces termination with an internal depth bound and, therefore, no cycle detection is activated. In this case, a warning will be printed and the solutions up to the internal depth bound are returned.

```
Maude> unify in UNIFICATION-EX4 :
   X:NList : X:NList : X:NList =? Y:NList : Y:NList : Z:NList : Y:NList .
Warning: Unification modulo the theory of operator _:_ has encountered
an instance for which it may not be complete.

Unifier 1
X:NList --> #1:NList : #1:NList : #1:NList : #1:NList
Y:NList --> #1:NList : #1:NList : #1:NList
Z:NList --> #1:NList : #1:NList : #1:NList

Unifier 2
X:NList --> #1:NList : #1:NList : #1:NList
Y:NList --> #1:NList : #1:NList
Z:NList --> #1:NList : #1:NList : #1:NList

Unifier 3
X:NList --> #1:NList : #1:NList
```

```
Y:NList --> #1:NList
Z:NList --> #1:NList : #1:NList : #1:NList
Warning: Some unifiers may have been missed due to incomplete
unification algorithm(s).
```

If the verbose mode is activated (see Section 23.15), Maude will print different internal messages associated to the situations above:

- Associative unification using cycle detection.
- Associative unification algorithm detected an infinite family of unifiers.
- Associative unification using depth bound of 5.
- Associative unification algorithm hit depth bound.

13.4.7 Associative with identity (AU) unification examples

Unification modulo associativity and identity was the missing case for associativity without commutativity. It has been implemented in Maude 3.1. Unification modulo associativity but with left identity or right identity is still missing and will be available in the future. However, it is currently supported in a different way, namely, by variant-based unification using an explicit equation (see Section 14.8). Note that unification modulo associativity and identity is, in general, not finitary, as in the case of only associativity, and the same conventions and warnings apply (see Section 13.4.6). Let us illustrate the use of the unification command in the presence of AU symbols using previous examples.

Consider a very simple module, adapting the previous theory `UNIFICATION-EX4`, with a binary associative symbol with an identity symbol for the empty list:

```
fmod UNIFICATION-EX5 is
  protecting NAT .
  sort NList .
  op nil : -> NList .
  subsort Nat < NList .
  op _:_ : NList NList -> NList [assoc id: nil] .
endfm
```

The unification problem below, using an associative symbol with identity, returns 32 unifiers using the `unify` command. Instead, the minimal set contains only 3, obtained by using the `irredundant unify` command.

```
Maude> irredundant unify in UNIFICATION-EX5 :
        X:NList : Y:NList : Z:NList =? P:NList : Q:NList .
Decision time: 2ms cpu (2ms real)

Unifier 1
X:NList --> #3:NList : #4:NList
Y:NList --> #1:NList
Z:NList --> #2:NList
P:NList --> #3:NList
Q:NList --> #4:NList : #1:NList : #2:NList

Unifier 2
X:NList --> #1:NList
```

```
Y:NList --> #3:NList : #4:NList
Z:NList --> #2:NList
P:NList --> #1:NList : #3:NList
Q:NList --> #4:NList : #2:NList

Unifier 3
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #4:NList : #3:NList
P:NList --> #1:NList : #2:NList : #4:NList
Q:NList --> #3:NList
```

This is an example where associativity-identity has fewer most general unifiers than associativity; see the unification command for theory `UNIFICATION-EX4` in page 332 above with 5 most general unifiers (both the `unify` and `irredundant unify` commands return 5 although only the former is shown).

The reader may be aware that the algorithm for associative-identity unification produces many redundant unifiers due to its collapse nature and how the order-sorted information is actually treated. Therefore, the use of the `irredundant unify` command is advised.

13.5 Some applications of unification

In this section we review briefly some applications that can be developed using a unification infrastructure like the one described in this chapter. We begin in Section 13.5.1 by discussing narrowing and narrowing-based unification algorithms. These algorithms are actually implemented in Core Maude's C++ and described in Chapter 14. We then explain in Section 13.5.2 how narrowing modulo an equational theory can be used for reachability analysis of concurrent systems described by rewrite theories, and, more generally, for symbolic temporal logic model checking of such systems. This narrowing-based reachability analysis is implemented in Core Maude's C++ and described in Chapter 15. Finally, we discuss briefly other automated deduction applications, including theorem proving ones.

13.5.1 Narrowing-based unification

If we have a dedicated algorithm (as the one supported by Maude) to solve unification problems in an order-sorted theory (Σ, Ax), then we can use it as a component to obtain a unification algorithm for theories of the form $(\Sigma, E \cup Ax)$, provided the equations E are unconditional, coherent, confluent and terminating modulo Ax [85].

The technique used under such conditions to obtain an $E \cup Ax$-unification algorithm from an Ax-unification algorithm is called *narrowing*, and is the obvious generalization of term rewriting to handle logical variables and perform a kind of symbolic execution. In ordinary term rewriting, if we want to apply a rewrite rule, say $l \to r$, to a term t at position p, the subterm $t|_p$ must be an *instance* of the lefthand side l, that is, there must be a substitution σ such that $t|_p = \sigma(l)$. Instead, in narrowing we can apply the rule $l \to r$ at a non-variable position p in t, provided the unification problem $t|_p =^? l$ (where the variables of l and t are assumed disjoint) has a nonempty set of unifiers. For any such unifier θ we then *narrow* the original term t to the substitution instance under θ of $t[r]_p$. We then write

$$t \rightsquigarrow \theta(t[r]_p)$$

for such a narrowing step. For example, in the standard, unsorted specification of the natural numbers, we can use the equation $x + s(y) = s(x + y)$ as a rewrite rule to narrow the term

$x' * (y' + z')$ at position 2 with substitution $\theta = \{x \mapsto y'', y \mapsto z'', y' \mapsto y'', z' \mapsto s(z'')\}$ to get the narrowing step

$$x' * (y' + z') \leadsto x' * s(y'' + z'').$$

In this example, θ is the most general unifier for the syntactic unification problem $y' + z' =^? x + s(y)$. However, in the same way as we can perform rewriting modulo a set of axioms Ax if we have an Ax-matching algorithm, we can likewise perform *narrowing modulo a set Ax of axioms* if we have an Ax-unification algorithm. That is, the unification problems $t|_p =^? l$ are now solved, not by syntactic unification, but by Ax-unification.

If a theory $(\Sigma, E \cup Ax)$ satisfies the above coherence, confluence, and termination modulo Ax requirements, we can systematically reduce $E \cup Ax$-unification problems to narrowing problems as follows:

1. we add a fresh new sort `Truth` to Σ with a constant `tt`;
2. for each top sort of each connected component of sorts we add a binary predicate `eq` of sort `Truth` and add to E the equation `eq(x,x) = tt`, where `x` has such a top sort;
3. we then reduce an $E \cup Ax$-unification problem $t =^? t'$ to the narrowing reachability problem

   $$\texttt{eq}(t, t') \leadsto^* \texttt{tt}$$

   modulo Ax in the theory extending $(\Sigma, E \cup Ax)$ with these new sorts, operators, and equations, where E and the new equations are used as rewrite rules.

That is, we search for all narrowing paths modulo Ax from $\texttt{eq}(t, t')$ to `tt`. Each such path then gives us a unifier of the equation $t =^? t'$, just by composing the unifiers of each narrowing step in the path. As explained in [17], narrowing can be performed not only in order-sorted equational theories, but also in membership equational theories, although the membership case is not yet implemented in Maude.

The just-described computation of $E \cup Ax$-unifiers by narrowing modulo Ax yields a complete but in general infinite set of $E \cup Ax$-unifiers. For the case when $Ax = \emptyset$, some sufficient conditions are known ensuring termination of the basic narrowing strategy (see, e.g., [84, 5, 3]), and therefore ensuring that the complete set of $E \cup Ax$-unifiers computed by basic narrowing is finite. However, for commonly occurring sets Ax of axioms, such as associativity-commutativity (AC), it is well-known that narrowing modulo AC "almost never terminates" and, furthermore, that narrowing strategies facilitating termination such as basic narrowing are incomplete [137, 36]. Based on the idea of "variants" in [36], a complete, yet quite efficient in terms of its search space, narrowing strategy modulo Ax called *folding variant narrowing* has been proposed in [70, 71]. Furthermore, in [69, 71, 23] sufficient checkable conditions on $(\Sigma, E \cup Ax)$ have been given ensuring that the $E \cup Ax$-unification algorithm provided by folding variant narrowing modulo Ax is finitary.

In Maude 2.6, a narrowing library developed by Santiago Escobar implemented the folding variant narrowing as a component, making $E \cup Ax$-unification available as part of Full Maude. Instead, $E \cup Ax$-unification is implemented in Core Maude's C++ level starting from Maude 2.7 (see Section 14.8).

13.5.2 Symbolic reachability analysis in rewrite theories

A rewrite theory[6], say $\mathcal{R} = (\Sigma, E \cup Ax, R)$, specified in Maude as a system module, describes a concurrent system whose states are $E \cup Ax$-equivalence classes of ground terms, and whose local

[6] All we say here applies also to rewrite theories with an additional freezing function ϕ specifying which arguments of each function symbol are *frozen*.

concurrent transitions are specified by the rules R. When formally analyzing the properties of $\mathcal{R}$, an important problem is ascertaining for specific patterns t and t' the following *symbolic reachability problem:*

$$\exists X\ t \longrightarrow^* t'$$

with X the set of variables appearing in t and t', which for this discussion we may assume are a disjoint union of those in t and those in t'. That is, t and t' symbolically describe sets of concurrent states $[\![t]\!]$ and $[\![t']\!]$ (namely, all the ground substitution instances of t, resp. t', or, more precisely, the $E \cup Ax$-equivalence classes associated to such ground instances). And we are asking: is there a state in $[\![t]\!]$ from which we can *reach* a state in $[\![t']\!]$ after a finite number of rewriting steps?

For example, $\mathcal{R}$ may specify a cryptographic protocol, t may symbolically describe a set of initial states, and t' may likewise describe a set of *attack states*. Then, if the above reachability question can be answered in the affirmative, the protocol $\mathcal{R}$ is insecure against the kinds of attacks described by t'. Furthermore, if the way of answering the reachability question is somehow constructive, we should be able to exhibit a concrete attack as a rewrite sequence violating the security of the protocol.

As explained in [114] and generalized in [110], provided the rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, R)$ is *topmost* (that is, all rewrites take place at the root of a term), or, as in the case of AC rewriting of object-oriented systems, $\mathcal{R}$ is "essentially topmost," and the rules R are coherent with E modulo Ax, narrowing with the rules R modulo the equations $E \cup Ax$ gives a constructive, sound, and complete method to solve reachability problems of the form $\exists X\ t \longrightarrow^* t'$, that is, such a problem has an affirmative answer if and only if we can find a finite narrowing sequence modulo $E \cup Ax$ of the form $t \rightsquigarrow^* \theta(t')$ for some θ. The method is *constructive*, because instantiating t with the composition of the unifiers for each step in the narrowing sequence gives us a concrete rewrite sequence witnessing the existential formula.

Of course, narrowing with R modulo $E \cup Ax$ requires performing $E \cup Ax$-unification at each narrowing step. As explained in Section 13.5.1, $E \cup Ax$-unification can itself be performed by narrowing with the equations E modulo Ax, provided E is coherent, confluent, and terminating modulo Ax. Therefore, in performing symbolic reachability analysis in a rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, R)$ there are usually *two* levels of narrowing and *two* levels of unification: narrowing with R modulo $E \cup Ax$ for reachability, and narrowing with E modulo Ax for unification purposes. Similarly, unification modulo $E \cup Ax$ is performed by narrowing, while unification modulo Ax is usually performed in a built-in way.

This is exactly the approach taken in the Maude-NPA protocol analyzer [66, 67], where cryptographic protocols are formally specified as rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup Ax, R)$, and the formal reachability analysis is performed in a *backwards* way, from an attack state to an initial state. This just means that we perform standard (forwards) reachability analysis with the rewrite theory $\mathcal{R}^{-1} = (\Sigma, E \cup Ax, R^{-1})$, where $R^{-1} = \{r \longrightarrow l \mid (l \longrightarrow r) \in R\}$. The equational theory $E \cup Ax$ typically specifies the algebraic properties of the cryptographic functions used in the given protocol, for example, public key encryption and decryption, exclusive or, modular exponentiation, and so on. Reasoning modulo such algebraic properties is very important to gain high levels of assurance, since it is well-known that some cryptographic protocols that can be proved secure under the standard Dolev-Yao model, in which the cryptographic functions are treated as a "black box," can actually be broken by an attacker that makes clever use of the algebraic properties of the cryptographic functions of the protocol. Besides using narrowing with rules modulo equations, the Maude-NPA tool uses several state space reduction techniques, including grammars that can describe sets of unreachable states that need not be explored [66], to drastically reduce the narrowing search space, often from an infinite set of states to a finite set of them, so that finite failure to find an attack becomes an actual proof of security.

Given a rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, R)$, we may be interested in verifying properties more general than existential questions of the form $\exists X\ t \longrightarrow^* t'$. Note that we can view such questions as questions about the *violation of an invariant*, because we can regard the set of states $[\![t']\!]$ as the *complement* of an invariant set of states, say I, which can be easily specified by an equationally-defined predicate. That is, proving the existential formula $\exists X\ t \longrightarrow^* t'$ is the same thing as finding a counterexample for the assertion $\mathcal{R}, t \models \Box I$. This is just a temporal logic satisfaction assertion (see Chapter 12), but with the following nonstandard features: (i) the term t does not describe a single initial state, but a possibly infinite set $[\![t]\!]$ of initial states; and (ii) there is no guarantee that the set of reachable states is finite. Therefore, standard model-checking techniques may not be usable, because of a possible double infinity: in the number of initial states, and in the number of states reachable for each of those initial states. One can also generalize the above reachability question $\mathcal{R}, t \models \Box I$ to questions of the form $\mathcal{R}, t \models \varphi$, with φ a temporal logic formula. The papers [68, 8] show how narrowing can be used (again, both at the level of transitions with rules R and at the level of equations E) to perform *logical model checking* to verify such temporal logic formulas; this is a a kind of *symbolic model checking* not in the binary decision diagram sense of "symbolic," which still remains finite-state, but in a much more general sense in which possibly infinite sets of states are finitely described by patterns with logical variables.

In Maude 2.7, a narrowing-based reachability analysis developed by Santiago Escobar was implemented as part of Full Maude. In Maude 3.0, a narrowing-based reachability analysis has been directly implemented in Core Maude's C++ (see Chapter 15).

13.5.3 Other automated deduction applications

The automated deduction application *par excellence*, and the one that historically, thanks to Alan Robinson, gave rise to the unification notion is *resolution-based theorem proving* [127]. Subsequent work by Gordon Plotkin [124] made it clear that not just syntactic unification, but unification *modulo* a set of equational axioms Ax is a very useful mechanism supporting theorem proving. Indeed, state-of-the-art resolution-based theorem provers routinely support unification modulo commonly occurring equational theories such as AC. Of course, the use of equational unification need not be restricted to resolution-based theorem provers. For example, the paper [129] shows how narrowing with sequent rules and equational unification can be used in a sequent-based theorem prover in which one can reason *modulo* both the equivalences given by the structural rules for sequents and also Boolean equivalences between formulas.

Yet another important application area is that of formal reasoning methods such as Knuth-Bendix equational completion (and its associated "inductionless induction" theorem-proving methods), checking local confluence of rewrite rules, and checking coherence of a set of rewrite rules with respect to a set of equations [138]. In all these formal reasoning methods one needs to compute *critical pairs* by unification of a term with a subterm of another term. In particular, tools such as the Maude Church-Rosser Checker (CRC) and Coherence Checker (ChC) were before restricted to theories where the only equational axiom supported was commutativity. The present built-in support for unification modulo a wide set Ax of axioms (further extensible to identity axioms by narrowing as explained in Section 13.5.1) has made it possible to have much more general versions of the Maude Church-Rosser Checker and Coherence Checker tools to reason about the confluence and coherence of Maude specifications modulo equational axioms specified as equational attributes in Maude modules and theories [55, 54, 56].

13.6 Endogenous vs. exogenous order-sorted unification algorithms

The current Maude order-sorted unification algorithm modulo axioms Ax is what we might call an *endogenous* algorithm, in the sense that the computation of order-sorted unifiers is intimately integrated with the order-sorted reasoning process, so that unifiers that do not type under the order-sorted typing restrictions are never generated. This makes such an algorithm typically more efficient, because the order-sorted typing restrictions may drastically cut the number of generated unifiers, particularly modulo axioms such as AC where the number of unsorted unifiers can be very large. That is, order-sorted unification, even though it lacks the unitary property of unsorted syntactic unification and is in general more expensive than unsorted unification in the syntactic case, can often be more efficient in the modulo Ax case because of the drastic reductions that can be achieved by order-sorted typing restrictions in the number of Ax-unifiers. Moreover, even in the syntactic case, the efficiency of deductive processes that use order-sorted unification can substantially increase, because order-sorted unification will *fail* more often than unsorted unification, leading to smaller search spaces.

However, from the early papers on order-sorted unification such as, e.g., [130, 112, 132] a more modular, although typically less efficient, approach to order-sorted unification, which we might call *exogenous* has been known. The basic idea is to *reuse* an existing unsorted unification algorithm modulo some axioms Ax (under some conditions on Ax) to compute order-sorted Ax-unifiers in the following way:

1. type information is removed from the order-sorted Ax-unification problem to convert it into an unsorted Ax-unification problem;

2. a complete set of unsorted Ax-unifiers is computed; and

3. the order-sorted Ax-unifiers of the original problem are obtained from the unsorted ones by a process of *filtering* the unsorted unifiers through an order-sorted reasoning process, in which the sorts of the variables in the original problem are taken into account. Each order-sorted unifier thus obtained is always a *specialization* of a corresponding unsorted one, where the unsorted variables have been specialized to given sorts; however, some unsorted unifiers, perhaps many, may be filtered out by this process and have no corresponding order-sorted unifiers.

For a state-of-the art study of the exogenous approach, allowing very general axioms Ax and proving the correctness of an order-sorted inference system to generate the order-sorted unifiers from the unsorted ones, see [82].

Both the endogenous and the exogenous approaches have their own advantages and disadvantages. The endogenous approach is more efficient, but it requires dedicated algorithms and implementations, so that unsorted unification algorithms and tools cannot be reused. The exogenous algorithms are less efficient because: (i) they can generate many unifiers that may later be discarded; (ii) a separate order-sorted filtering process is needed; and (iii) changes of representation, and even parsing, are required between unsorted and order-sorted representations (particularly when existing unsorted algorithms are reused). However, they are more modular and flexible, so that one can with relatively little effort obtain an order-sorted unification algorithm from an unsorted one.

In Maude we have experimented with, and benefited from, both an exogenous algorithm and the current endogenous one. The exogenous algorithm was developed in collaboration with Evelyn Contejean and Claude Marché from Université Paris-Sud, and involved also the efforts of Prasanna Thati and Joe Hendrix at UIUC. It reused the rich library of unsorted unification

algorithms modulo axioms of the CiME system [38], which could be called from Maude in an experimental version. Inside Maude, it used the order-sorted inference system to compute order-sorted unifiers developed by Joe Hendrix and described in [82].

This exogenous algorithm has been extensively used in a previous version of the Maude-NPA tool, and has been shown effective in finding attacks to cryptographic protocols modulo nontrivial equational theories of the form $E \cup Ax$ [65]. The exogenous algorithm has also been extremely useful in *testing* the endogenous one. Because of the large number of unifiers generated and the complex nature of semantic unification algorithms, their testing is a nontrivial matter, and the automation of such testing is quite difficult. Thanks to the exogenous algorithm, and through the efforts of Ralf Sasse and Santiago Escobar, it has been possible to generate large numbers of random unification problems of different sizes in which the sets of unifiers generated by the exogenous and endogenous order-sorted unification algorithms have been automatically compared. This testing uncovered several bugs in an earlier alpha version of the Maude endogenous algorithm, and has also served to evaluate in practice the greater efficiency of the endogenous algorithm developed by Steven Eker.

13.7 Some notes on the implementation of unification

Order-sorted unification is NP-complete in general because Boolean algebra can be encoded as an order-sorted free theory signature and hence satisfiability can be reduced to an order-sorted free theory unification problem. In practice, reasonable performance can be obtained using a binary decision diagram technique to compute sorts for free variables occurring in unsorted unifiers. Furthermore in the AC case, sort information can be pushed into the unsorted unification algorithm and used to prune the Diophantine basis and the choice of subsets drawn from such a basis [60].

The unification theory combination framework and AC-unification algorithm are based on [16], while the Diophantine system solver used by the AC algorithm is based on [37]. The unification algorithm has been thoroughly tested by Santiago Escobar and Ralf Sasse using CiME [38] as an oracle, and has shown better average performance than CiME on the same problems.

The addition of ACU to the theories handled by the dedicated unification algorithm in Maude required substantial changes to the unification infrastructure implemented in previous versions of Maude for C and AC theories because of the problems associated with collapse theories. In this section we give an overview of the techniques used and highlight a novel algorithm for selecting sets of Diophantine basis elements during the computation of ACU unifiers.

13.7.1 Combining unification algorithms

The basic approach to solving unification problems where function symbols are drawn from more than one theory is variable abstraction where *alien subterms*, i.e., subterms headed by a symbol from a theory different from that of the top symbol of the parent term, are replaced by fresh variables to form *pure* unification subproblems which only involve variables and function symbols from a single theory and which can be passed to a unification algorithm for such a theory. Proving termination of combinations of algorithms is nontrivial, as variables are necessarily shared between theories and the unification of variables in one theory can create new unification subproblems in another theory, potentially ad infinitum. Stickel's algorithm [133], which combined the AC and free theories, required an elaborate termination proof by Fages [72]. Boudet et al. [16] proposed a much simpler approach where all unification subproblems

and variable bindings in a given theory are solved (and re-solved if another subproblem in that theory is created) simultaneously. This method requires a simultaneous E-unification algorithm for each theory E and was the method implemented in Maude for C, AC, and $\emptyset$ prior to the addition of ACU.

Collapse theories add two major complications to the combination of unification algorithms. Firstly, theory clashes where two terms with top symbols from different theories are required to unify can no longer be treated as a failure, since if one of the top symbols belongs to a collapse theory, a collapse may occur, yielding solutions. Secondly, *compound cycles*, that is, problems of the form $x_1 =^? t_1(\ldots, x_2, \ldots), x_2 =^? t_2(\ldots, x_3, \ldots), \ldots, x_n =^? t_n(\ldots, x_1, \ldots)$ where the terms t_i are pure in different theories, can no longer be treated as failure, since solutions may be possible via collapse.

Several authors have proposed combination schemes that can handle collapse theories. We use a simplified version of an algorithm due to Boudet [15]. The original algorithm also handles nonregular theories but we omit that capability to simplify the implementation. The key idea is that each theory E needs a restricted simultaneous E-unification algorithm which solves the simultaneous unification problem for pure equations that are pure in E but where certain variables may be marked as only being allowed to unify with other variables. A theory clash subproblem $f(\ldots) =^? g(\ldots)$, is split into a disjunction of two subproblems each of which is a conjunction $x =^? f(\ldots) \wedge x =^? g(\ldots)$ where x is a fresh variable. In one subproblem x is marked in the f equation and in the other subproblem x is marked in the g equation; either or both branches of the search may return solutions. Restricted unification is also used to break compound cycles. Because we do not handle nonregular theories, Boudet-style variable-elimination algorithms are unnecessary.

Boudet's algorithm assumes that theories are disjoint, i.e., that they do not share function symbols. Because in Maude this is not quite true — identities can contain symbols from other theories — we need to handle a special kind of variable elimination. We illustrate the issue with the following example:

```
fmod UNIFICATION-CYCLE is
  sort S .
  vars X Y : S .
  ops a b c d : -> S .
  op f : S S -> S [assoc comm id: g(c, d)] .
  op g : S S -> S [assoc comm id: f(a, b)] .
endfm

Maude> unify X =? f(Y, a, b) /\ Y =? g(X, c, d) .
```

Here the unification problem would already be in solved form but for the compound cycle formed by the `X` and `Y` variables. Restricted unification cannot break this cycle, since neither of the righthand sides can collapse out of their theory. However, putting `Y = g(c, d)` eliminates `Y` from the first equation yielding `X = f(a, b)` which eliminates `X` from the second equation, thus yielding a solution. This situation is somewhat pathological in Maude programs, and we do not really care about performance in its handling. Maude handles it by looking for this kind of cyclic dependency between theories when the signature is preprocessed and setting a flag so that a brute force variable elimination algorithm will be used to try and break compound cycles at unification time.

13.7.2 Combining incomplete unification algorithms

The unification infrastructure now supports the notion of incomplete unification algorithms (see Section 13.4.6). This has several major components. Firstly the algorithm for combining uni-

fication algorithms now prioritizes theories that have complete algorithms and amongst those, prioritizes theories that are expected to have fewer partial solutions. The idea of this heuristic is to minimize branching and when a unification problem with an incomplete unification algorithm has to be solved, to have more information on shared variables (ideally they would be bound to terms in collapse-free theories), in order to minimize the likelihood of encountering an incomplete case. Secondly, when different equations on the same theory are given as a unification problem, equations with a lower likelihood of encountering an incomplete case are prioritized. Thirdly, when incompleteness does occur it is tracked through the various levels of unification calls so that appropriate warnings can be issued.

13.7.3 Diophantine basis element selection

We solve restricted simultaneous ACU unification using an extension of the simultaneous AC unification algorithm in [16]. For an ACU function symbol f we are presented with a set of flattened pure equations that take the form $f(x_1^{p_1}, \ldots, x_n^{p_n}) =^? f(y_1^{q_1}, \ldots, y_m^{q_m})$ or $x_1 =^? f(y_1^{q_1}, \ldots, y_m^{q_m})$. Each f-equation yields a Diophantine equation $p_1X_1 + \cdots + p_nX_n = q_1Y_1 + \cdots + q_mY_m$ or respectively, $X_1 = q_1Y_1 + \cdots + q_mY_m$ where the X_i's and Y_i's are non-negative Diophantine variables. If an original variable is marked in some equation, the corresponding Diophantine variable receives an upper-bound of 1. Also, we may be able to obtain an upper-bound from order-sorting information, using the signature analysis technique in [57].

The general solution to a set of non-negative Diophantine equations is a set of basis elements from which all solutions can be obtained by linear combination. Upper-bound information may trivially eliminate some basis elements from consideration and can be used by the Diophantine solver to terminate the search for basis elements early.

A fresh variable z_k is allocated for each basis element α_k and unifiers are formed by finding sets of basis elements that satisfy certain properties and constructing assignments $x_i \leftarrow f(\ldots, z_k^{\alpha_{k,i}}, \ldots)$ where k ranges over the indices of the selected basis elements and $\alpha_{k,i}$ is the value of X_i in the basis element α_k.

The criteria for choosing the sets of basis elements is the key difference between AC unification, ACU unification, and restricted ACU unification. With AC unification, every selection of basis elements whose sum yields a nonzero value for each X_i and Y_i must be considered. With ACU unification that requirement is lifted because of the availability of an identity element. The identity element also means that any assignment including basis element α_k generalizes the same assignment with α_k removed by assigning the identity element to z_k and thus there is a single most general solution, formed by selecting all the basis elements.

In the case of restricted ACU unification, we may have upper-bounds on variables because they are marked. In Maude, order-sorted considerations may place upper-bounds on variables, and may also place a lower-bound of 1 on variables where the corresponding original variable has a sort that cannot take the identity element. In order to find a complete set of unifiers we need to find all maximal sets of basis elements whose sum satisfies the upper and lower-bounds on the variables.

Several explicit schemes for searching the subsets of basis elements were tried but the search was typically the dominant cost for ACU unification, often rendering the solution of quite modest unification problems impractical. In the current implementation this search is performed symbolically using a Binary Decision Diagram (BDD) [20] based algorithm. A BDD variable is allocated for each basis element, whose value, true or false, denotes whether the basis element is included in the subset. A BDD, called *legal*, is constructed, which evaluates to true on exactly those valuations that correspond to selections of basis elements that satisfy the upper- and lower-bound constraints on each Diophantine variable. Enforcement of the upper-bounds on the sum is done using dynamic programming and the BDD *ite* operation. Using the BDD *legal*,

a second BDD, called *maximal*, is constructed which is true on exactly those valuations where *legal* is true, and changing a false into a true makes *legal* false. These valuations of the BDD variables and thus the subsets of basis elements they encode are then recovered by tracing the paths from the root to the true terminal in *maximal*. This method yielded a dramatic speed up (from hours to milliseconds) on problems of useful size.

Chapter 14

Variants and Variant Unification

14.1 Introduction

As explained in Section 13.2, Maude features order-sorted unification modulo axioms Ax, including associativity[1] (A), commutativity (C), associative-identity (AU)[1], commutativity-identity (CU), associativity-commutativity (AC), associativity-commutativity-identity (ACU), and (left-, right- or two-sided) identity (Ul, Ur, U). However, order-sorted equational unification in full generality considers a decomposition of an equational theory $(\Sigma, E \cup Ax)$ into *two* problems: one of Ax-unification, and another of $E \cup Ax$-unification that uses an Ax-unification algorithm as a subroutine. As explained in Section 13.5.1, algorithms for $E \cup Ax$-unification have been extensively defined by using narrowing-based unification, where for a unification problem $t =^? t'$ we obtain the search space associated to narrowing the term $\mathtt{eq}(t, t')$ using E modulo Ax and search for all paths from $\mathtt{eq}(t, t')$ to the truth constant `tt`. However, we use the notion of *variants* of a term for generating such a narrowing search space.

14.2 Term variants

Comon-Lundh and Delaune's notion of *variant* [36] characterizes the instances of a term w.r.t. an equational theory $E \cup Ax$ such that the equations E are confluent, terminating, and coherent modulo axioms Ax.

> The E, Ax-*variants* of a term t are *pairs* (t', θ), with θ a substitution and t' the E, Ax-canonical form of $\theta(t)$.

A preorder relation of generalization that holds between such pairs can be given: we say a variant (t_1, θ_1) of a term t is more general than another variant (t_2, θ_2) of the same term t if there is a substitution ρ such that $\rho(t_1) =_{Ax} t_2$, and $(\theta_1; \rho)|_{Var(t)} =_{Ax} (\theta_2)|_{Var(t)}$. A *complete set of* E, Ax-*variants* (up to renaming) of a term t is a subset V of E, Ax-variants of t such that for any variant $(\sigma(t)\downarrow_{E,Ax}, \sigma)$ there exists a more general variant (t', θ) in V.

In order to avoid clashing of algorithms and notions, we have decided that the equations used for variant generation (and variant-based unification) should be identifiable and clearly distinguished from standard equations in Maude. For this purpose we have defined a new

[1] As already explained, for a useful subclass of associative and associative-identity unification problems complete sets of unifiers are returned, and in all other cases a possibly incomplete such set is returned with a warning. See Sections 13.4.6 and 13.4.7.

attribute for equations: the keyword `variant`. This implies that if the user wants to use an equation `t = t'` both for variant generation and for simplification, it should be duplicated: `eq t = t' .` and `eq t = t' [variant] .` No equation with the variant attribute can have the `owise` attribute. Note that what this allows is a greater flexibility at the operational level when combining variant generation and simplification: by the above method, an equation can be used for either purpose (declared only once in the appropriate way), or for both, by a double declaration.

For example, consider the following functional module defining the addition function `_+_` on natural numbers built from `0` and `s`:

```
fmod NAT-VARIANT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars X Y : Nat .
  eq [base] : 0 + Y = Y [variant] .
  eq [ind] : s(X) + Y = s(X + Y) [variant] .
endfm
```

The term `X + s(0)` has an infinite number of variants w.r.t. those equations, i.e.,

- $(\texttt{s(0)}, \{\texttt{X} \mapsto \texttt{0}\})$,
- $(\texttt{s(s(Y))}, \{\texttt{X} \mapsto \texttt{s(Y)}\})$,
- $(\texttt{s(s(0))}, \{\texttt{X} \mapsto \texttt{s(0)}\})$,
- $(\texttt{s(s(s(Y)))}, \{\texttt{X} \mapsto \texttt{s(s(Y))}\})$,
- $(\texttt{s(s(s(0)))}, \{\texttt{X} \mapsto \texttt{s(s(0))}\})$, ...

Indeed, there is no finite, complete, most general set of variants for that term. However, the term `0 + X` has a finite number of most general variants w.r.t. those equations, i.e., $(\texttt{X}, id)$. Obviously, there are many more variants, such as $(\texttt{0}, \{\texttt{X} \mapsto \texttt{0}\})$, but they are all instances of the most general one.

An equational theory $E \cup Ax$ has the *finite variant property* iff there is a finite complete set of most general variants for each term. This property also ensures the existence of a generic *finitary* $E \cup Ax$-unification algorithm based on computing variants, as shown in Section 14.8. However, if a theory does not have the finite variant property, we are still able to incrementally enumerate all the variants of a term, as explained below in Section 14.6.

Consider the following equational theory for exclusive or.

```
fmod EXCLUSIVE-OR is
  sorts Nat NatSet .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  subsort Nat < NatSet .
  op mt : -> NatSet [ctor] .
  op _*_ : NatSet NatSet -> NatSet [ctor assoc comm] .

  vars X Y Z W : [NatSet] .
  eq [idem] :   X * X = mt   [variant] .
  eq [idem-Coh] : X * X * Z = Z [variant] .
```

```
  eq [id] :      X * mt = X   [variant] .
endfm
```

This theory has the finite variant property, as proved manually in [36] and automatically in [69]. For instance, the term `X * X` has a finite set of most general variants, just $(\texttt{mt}, id)$. And the term `X * Y` has also a finite, complete set of most general variants:

1. $(\texttt{X * Y}, id)$,
2. $(\texttt{Z}, \{\texttt{X} \mapsto \texttt{mt}, \texttt{Y} \mapsto \texttt{Z}\})$,
3. $(\texttt{Z}, \{\texttt{X} \mapsto \texttt{Z}, \texttt{Y} \mapsto \texttt{mt}\})$,
4. $(\texttt{Z}, \{\texttt{X} \mapsto \texttt{Z * U}, \texttt{Y} \mapsto \texttt{U}\})$,
5. $(\texttt{Z}, \{\texttt{X} \mapsto \texttt{U}, \texttt{Y} \mapsto \texttt{Z * U}\})$,
6. $(\texttt{mt}, \{\texttt{X} \mapsto \texttt{U}, \texttt{Y} \mapsto \texttt{U}\})$, and
7. $(\texttt{Z}_1 \texttt{ * } \texttt{Z}_2, \{\texttt{X} \mapsto \texttt{U * Z}_1, \texttt{Y} \mapsto \texttt{U * Z}_2\})$

Note that if variable `X` in the equational theory is changed[2] from sort `[NatSet]` to `Nat`, then the theory does not have the finite variant property, since every pair of similar elements has to be separately eliminated, whereas now chunks of similar elements can be eliminated at once. Also, note that the symbol `_*_` cannot be made ACU instead of AC, because then the equation[3] `X * X * Z = Z` is not ACU-terminating.

The finite variant property happens to be an undecidable problem [14]. However, a semi-decision procedure for checking the finite variant property has been developed which works well in practice: it has recently [23] been shown that, in order to prove the finite variant property for an equational theory $(\Sigma, E \cup Ax)$, it is enough to check, for each function symbol $f \in \Sigma$, whether or not each pattern of the form $f(X_1, \ldots, X_n)$ has a finite number of variants, where the X_i are distinct variables of the appropriate kind and n is the arity of f. This can be done by attempting to generate all the variants of $f(X_1, \ldots, X_n)$ as described in Section 14.4 below.

Variants are used for variant-based unification in Section 14.8, and such a variant-based unification is later used in Section 15.6 for symbolic reachability analysis. Before defining variant-based unification, in Section 14.3 we introduce the class of equational theories admissible for variant generation, and thus for variant-based unification. We also provide in Section 14.4 a command `get variants` for user generation of variants.

14.3 Theories currently supported

The equational theories that are admissible for variant generation are as follows. Let `fmod` $(\Sigma, E \cup Ax)$ `endfm` (resp. `fth` $(\Sigma, E \cup Ax, R)$ `endfth`) be an order-sorted functional module (resp. functional theory) where E is a set of equations specified with the `eq` keyword and the attribute `variant`, and Ax is a set of axioms such that (Σ, Ax) satisfies the restrictions explained in Section 13.3. Furthermore, the equations E must satisfy the following extra conditions:

- The equations E are unconditional, confluent, terminating, sort-decreasing, and coherent modulo Ax (see also Section 21.5 for coherence details).
- An equation's lefthand side cannot be a variable, and the `owise` feature is not allowed.

[2] Nothing happens if `X` is changed from `[NatSet]` to `[Nat]`, since `NatSet` and `Nat` have the same kind.
[3] This is the only equation necessary for ACU-coherence and the other two would be eliminated.

Any system module `mod` $(\Sigma, G \cup E \cup Ax, R)$ `endm` (or system theory `th` $(\Sigma, G \cup E \cup Ax, R)$ `endth`), where G is an additional set of equations (without the `variant` attribute!) and R is a set of rules, is also considered admissible for variant generation if the equational part $(\Sigma, E \cup Ax)$ satisfies the conditions described above. Note that when an equational theory $(\Sigma, G \cup E \cup Ax)$ is entered into Maude, each equation in E (used for variant computation) must include the `variant` attribute. Note that equations in G do not have any restriction, i.e., they can be conditional equations, with the `owise` attribute, etc.

14.4 The `get variants` command

Given a module $\langle$*`ModId`*$\rangle$, Maude provides two variant generation commands of the form:

`get variants [` n `] in` $\langle ModId \rangle$ `:` $\langle Term \rangle$ `.`
`get irredundant variants [` n `] in` $\langle ModId \rangle$ `:` $\langle Term \rangle$ `.`

where n is an optional argument providing a bound on the number of variants requested, so that if the cardinality of the set of variants is greater than the specified bound, the variants beyond that bound are omitted; and, as usual, if no module is mentioned, the current module is used.

Maude allows an incremental generation of variants, as described in Section 14.6 below. When a theory does not have the finite variant property, an incremental generation of the (possibly infinite) set of most general variants would be returned by the first command `get variants`. However, the second command, `get irredundant variants`, is useful for theories that do have the finite variant property, since it will provide the set of most general variants of a term, which is the basis for variant-based unification in Section 14.8.

For example, we can check that the `EXCLUSIVE-OR` module above has the finite variant property by simply generating the variants for the exclusive-or symbol $*$.

```
Maude> get irredundant variants in EXCLUSIVE-OR : X * Y .

Variant 1
[NatSet]: #1:[NatSet] * #2:[NatSet]
X --> #1:[NatSet]
Y --> #2:[NatSet]

Variant 2
NatSet: mt
X --> %1:[NatSet]
Y --> %1:[NatSet]

Variant 3
[NatSet]: %1:[NatSet] * %3:[NatSet]
X --> %1:[NatSet] * %2:[NatSet]
Y --> %2:[NatSet] * %3:[NatSet]

Variant 4
[NatSet]: %1:[NatSet]
X --> %1:[NatSet] * %2:[NatSet]
Y --> %2:[NatSet]

Variant 5
[NatSet]: %2:[NatSet]
X --> %1:[NatSet]
```

```
Y --> %1:[NatSet] * %2:[NatSet]

Variant 6
[NatSet]: %1:[NatSet]
X --> mt
Y --> %1:[NatSet]

Variant 7
[NatSet]: %1:[NatSet]
X --> %1:[NatSet]
Y --> mt

No more variants.
```

The above example illustrates a difference between unifiers returned by the built-in unification modulo axioms and unifiers returned by variant generation or variant-based unification: there are two forms of fresh variables, the former *#n:Sort* and the new *%n:Sort*. The reasons for this distinction are immaterial: they are adopted conventions dictated by implementation choices. Both forms represent fresh variables and both share the same counter for new fresh variables. The user is required not to use variables of any of these two forms in submitted unification problems (either modulo axioms or variant-based). When variant-based unification is used at the metalevel (see Section 17.6.9), there are two possibilities: (i) a counter for new fresh variables must take into account the numbers used for both forms of fresh variables, or (ii) an alternative identifier for new variables can be given.

Recall that memberships are discarded completely. For instance, we can modify the previous example to include a membership definition for a new sort `Empty`.

```
fmod EXCLUSIVE-OR-MB is
  sorts Nat NatSet .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  subsort Nat < NatSet .
  op mt : -> NatSet [ctor] .
  op _*_ : NatSet NatSet -> NatSet [ctor assoc comm] .

  sort Empty .
  subsort Empty < NatSet .
  mb mt : Empty .

  vars X Y Z : [NatSet] .
  eq [idem] :    X * X = mt   [variant] .
  eq [idem-Coh] : X * X * Z = Z [variant] .
  eq [id] :      X * mt = X   [variant] .
endfm
```

We can ask then for the variants of the exclusive-or symbol $*$ when restricted to the sort `Empty`, and the results use the variant equations but not the membership information.

```
Maude> get irredundant variants in EXCLUSIVE-OR-MB : X:Empty * Y:Empty .

Variant 1
NatSet: #1:Empty * #2:Empty
X:Empty --> #1:Empty
Y:Empty --> #2:Empty
```

```
Variant 2
Empty: mt
X:Empty --> %1:Empty
Y:Empty --> %1:Empty

No more variants.
```

Note that the membership is used to compute the least sort of terms involved in the results (like the constant `mt` above), but is not used during variant generation. For example, this process is not able to instantiate any of the two variables to the constant `mt`.

Consider now the following version of the vending machine to buy apples (`a`) or cakes (`c`) with dollars (`$`) and/or quarters (`q`). The reader can check that the only difference with the `UNIF-VENDING-MACHINE` module in Section 13.4.4 is the `change` equation, where we have added the attribute `variant` and a variable `M` to make it *ACU*-coherent (see Section 21.5 for details on *ACU*-coherence).

```
mod VARIANT-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : < M $ > => < M c > .
  rl [buy-a] : < M $ > => < M a q > .
  eq [change] : q q q q M = $ M [variant] .
endm
```

Note that the `change` equation satisfies the finite variant property, as proved by generating the variants of symbol `__`.

```
Maude> get irredundant variants in VARIANT-VENDING-MACHINE : X:Marking Y:Marking .

Variant 1
State: #1:Marking #2:Marking
X:Marking --> #1:Marking
Y:Marking --> #2:Marking

Variant 2
State: %1:Marking %2:Marking
X:Marking --> q q q %1:Marking
Y:Marking --> q %2:Marking

Variant 3
State: %1:Marking %2:Marking
X:Marking --> q q %1:Marking
Y:Marking --> q q %2:Marking

Variant 4
State: %1:Marking %2:Marking
```

```
X:Marking --> q %1:Marking
Y:Marking --> q q q %2:Marking
```

We can also generate the variants of the state `< $ q q X:Marking >` containing at least a dollar and two quarters.

```
Maude> get irredundant variants in VARIANT-VENDING-MACHINE : < $ q q X:Marking > .

Variant 1
State: < $ q q #1:Marking >
X:Marking --> #1:Marking

Variant 2
State: < $ $ %1:Marking >
X:Marking --> q q %1:Marking
```

These two variants form a finite, complete, and most general set of variants for the given term; for example, the variant

```
{< $ $ q q Y:Marking >, X:Marking --> q q q q Y:Marking}
```

is an instance of the first variant above, i.e., the canonical form `< $ $ q q Y:Marking >` is an instance of the normal form `< $ q q #1:Marking >` of the first variant, and the (normalized version) of the instantiating substitution, i.e., `#1:Marking --> $ Y:Marking`, is an instance of the empty substitution of the first variant.

We can consider a more complex equational theory such as Lankford's formalization of an Abelian group, specified in the following module.

```
fmod ABELIAN-GROUP is
  sorts Element .
  op _+_ : Element Element -> Element [comm assoc prec 30] .
  op -_ : Element -> Element [prec 20] .
  op 0 : -> Element .
  vars X Y Z : Element .
  eq X + 0 = X [variant] .
  eq X + - X = 0 [variant] .
  eq X + - X + Y = Y [variant] .
  eq - - X = X [variant] .
  eq - 0 = 0 [variant] .
  eq - X + - Y = -(X + Y) [variant] .
  eq -(X + Y) + Y = - X [variant] .
  eq -(- X + Y) = X + - Y [variant] .
  eq - X + - Y + Z = -(X + Y) + Z [variant] .
  eq -(X + Y) + Y + Z = - X + Z [variant] .
endfm
```

The generation of the variants for the addition symbol takes more time and provides 47 variants:

```
Maude> get irredundant variants in ABELIAN-GROUP : X + Y .

Variant 1
Element: #1:Element + #2:Element
X --> #1:Element
Y --> #2:Element

Variant 2
Element: %1:Element
```

```
X --> 0
Y --> %1:Element
...

Variant 46
Element: %2:Element + - (%3:Element + %4:Element)
X --> %5:Element + - (%1:Element + %3:Element)
Y --> %1:Element + %2:Element + - (%4:Element + %5:Element)

Variant 47
Element: - (%2:Element + %3:Element)
X --> %4:Element + - (%1:Element + %2:Element)
Y --> %1:Element + - (%3:Element + %4:Element)
```

And the minus sign symbol has four variants:

```
Maude> get irredundant variants in ABELIAN-GROUP : - X .

Variant 1
Element: - #1:Element
X --> #1:Element

Variant 2
Element: %1:Element
X --> - %1:Element

Variant 3
Element: 0
X --> 0

Variant 4
Element: %1:Element + - %2:Element
X --> %2:Element + - %1:Element
```

14.5 Variant generation with irreducibility constraints

Variant generation is useful to many applications (see Sections 13.5 and 15.2) but especially for variant-based unification, as explained in Section 14.8 below. One special version of the variant generation algorithm implemented in Maude 3.0 includes irreducibility conditions, i.e., a term is not allowed to be reducible after applying a substitution. This is useful when, for example, the variants of a term may have been generated before and one is not interested in generating the variants of a variant of a term, since the variant t_3 of a variant t_2 of a term t_1 must be already covered by a variant of t_1 (maybe t_2 itself). The irreducibility condition is appended by using the keywords `such that T1, T2, T3, ... irreducible`.

For example, we may reconsider the generation of the set of irredundant variants of the expression `X * Y` in page 348 but now assuming that the term itself is irreducible, that is, no further variant must be computed.

```
Maude> get irredundant variants in EXCLUSIVE-OR : X * Y such that X * Y irreducible .

Variant 1
[NatSet]: #1:[NatSet] * #2:[NatSet]
X --> #1:[NatSet]
```

```
Y --> #2:[NatSet]
```

Similarly, we may reconsider the generation of the set of irredundant variants of the state `< $ q q X:Marking >` in page 351 but assuming that different subterms separated by commas are irreducible.

```
Maude> get irredundant variants in VARIANT-VENDING-MACHINE : < $ q q X:Marking >
such that q q X:Marking, q X:Marking, X:Marking irreducible .

Variant 1
State: < $ q q #1:Marking >
X:Marking --> #1:Marking
```

This command clearly discards the variant where `X:Marking` is mapped to `q q %1:Marking`, since it violates the condition that `q q X:Marking` must be irreducible under substitution.

14.6 Incremental variant generation

Another interesting feature is that variant generation is incremental and in this way we are able to give partial support to theories that do not have the finite variant property. Let us consider the functional module for addition `NAT-VARIANT` in Section 14.1 that does not have the finite variant property. On the one hand, it is possible to have a term with a finite number of most general variants although the theory does not have the finite variant property. For instance, the term `s(0) + X` is simplified into `s(X)`.

```
Maude> get variants in NAT-VARIANT : s(0) + X:Nat .

Variant 1
Nat: s(#1:Nat)
X:Nat --> #1:Nat
```

On the other hand, we can approximate the number of variants of a term that we suspect does not have a finite number of most general variants. For instance, the term `X + s(0)` has an infinite number of most general variants and we can approximate that infinite set of variants by including a bound in the command, as it is also done for unification modulo axioms (see Section 13.4.2).

```
Maude> get variants [10] in NAT-VARIANT : X:Nat + s(0) .

Variant 1
Nat: #1:Nat + s(0)
X:Nat --> #1:Nat

Variant 2
Nat: s(0)
X:Nat --> 0

...

Variant 10
Nat: s(s(s(s(s(0)))))
X:Nat --> s(s(s(s(0))))
```

Note that typing `get irredundant variants` in the previous command would force Maude to generate the infinite set of variants before collecting the ten most general ones.

This incremental variant generation may be useful in applications where obtaining a complete set of variants when possible is important. If we do not know a priori whether a term has a finite number of most general variants, we can incrementally increase the bound and if we obtain a number of variants smaller than the bound, we know for sure that it had a finite number of most general variants.

When an equational theory satisfies the requirements in Section 14.3 and does not have the finite variant property, it is because there are terms with an infinite set of most general variants. Of course, if the user does not provide a bound, Maude does not stop because it always returns a complete set of variants. However, when the user provides such a bound, the process will always terminate with Maude returning a finite set of variants. As said above, the number of returned variants can be smaller than the given bound when the term has indeed a finite set of variants, but it will coincide with the bound otherwise.

14.7 Variant generation in incomplete unification examples

The unification infrastructure now supports the notion of incomplete unification algorithms (see Section 13.4.6) and variant generation and variant unification have been updated.

Let us consider an equational theory combining variant equations and associativity.

```
fmod VARIANT-UNIFICATION-ASSOC is
  protecting NAT .
  sort NList .
  subsort Nat < NList .

  op _:_ : NList NList -> NList [assoc ctor] .

  var E : Nat .
  var L : NList .

  ops tail prefix : NList ~> NList .
  ops head last : NList ~> Nat .
  eq head(E : L) = E [variant] .
  eq tail(E : L) = L [variant] .
  eq prefix(L : E) = L [variant] .
  eq last(L : E) = E [variant] .

  op duplicate : NList ~> Bool .
  eq duplicate(L : L) = true [variant] .
endfm
```

Some terms have a finite set of most general variants modulo associativity.

```
Maude> get variants in VARIANT-UNIFICATION-ASSOC : head(prefix(tail(L))) .

Variant 1
Nat: head(prefix(tail(#1:NList)))
L --> #1:NList

Variant 2
Nat: head(prefix(%2:NList))
L --> %1:Nat : %2:NList
```

```
Variant 3
Nat: head(#2:NList)
L --> #1:Nat : #2:NList : #3:Nat

Variant 4
Nat: %3:Nat
L --> %1:Nat : %3:Nat : %4:NList : %2:Nat

No more variants.
```

However, some terms may hit incomplete associative unification calls (see Section 13.4.6), and an incompleteness warning for associative unification will be printed.

```
Maude> get variants in VARIANT-UNIFICATION-ASSOC :
         duplicate(prefix(L) : tail(L)) .

Variant 1
[Bool]: duplicate(prefix(#1:NList) : tail(#1:NList))
L --> #1:NList

Variant 2
[Bool]: duplicate(%1:NList : tail(%1:NList : %2:Nat))
L --> %1:NList : %2:Nat

Variant 3
[Bool]: duplicate(prefix(%1:Nat : %2:NList) : %2:NList)
L --> %1:Nat : %2:NList

Variant 4
[Bool]: duplicate(#1:Nat : #2:NList : #2:NList : #3:Nat)
L --> #1:Nat : #2:NList : #3:Nat

Variant 5
[Bool]: duplicate(#1:Nat : #2:Nat)
L --> #1:Nat : #2:Nat

Warning: Unification modulo the theory of operator _:_ has encountered
an instance for which it may not be complete.
Variant 6
Bool: true
L --> %1:Nat : %1:Nat : %1:Nat

Variant 7
Bool: true
L --> %1:Nat : %1:Nat

No more variants.
Warning: Some variants may have been missed due to incomplete unification algorithm(s).
```

Note that the term `duplicate(prefix(L) : tail(L))` has an infinite set of most general variants for the case of returning the variant term `true`, i.e., the family of substitutions $\{\texttt{L:NList} \mapsto \texttt{N:Nat} : \cdots : \texttt{N:Nat}\}$. This is due to the associative unification call $(\texttt{N:Nat} : \texttt{L:NList}) =^{?} (\texttt{L:NList} : \texttt{N:Nat})$ invoked internally by the variant generation process (see Section 13.4.6 for a similar associative unification problem with an infinite set of most general unifiers).

14.8 Variant-based equational order-sorted unification

The intimate connection between E, Ax-variants and $E \cup Ax$-unification is as follows. Suppose that we extend the equational theory $(\Sigma, E \cup Ax)$ to $(\widehat{\Sigma}, \widehat{E} \cup Ax)$ by adding to Σ a new sort `Truth`, not related to any sort in Σ, with a constant `tt`, and, for each top sort of a connected component `[s]`, an operator `eq : [s] [s] -> Truth`; and where $\widehat{E}$ is the result of adding for each top sort `[s]` an extra (oriented) equation `eq(x,x) = tt` (where `x` is a variable of sort `[s]`) to E. Then, given any two terms t, t', if θ is an $E \cup Ax$-unifier of t and t', then the E,Ax-canonical forms of $\theta(t)$ and $\theta(t')$ must be Ax-equal and therefore the pair $(\mathtt{tt}, \theta)$ must be a variant of the term $\mathtt{eq}(t, t')$. Furthermore, if the term $\mathtt{eq}(t, t')$ has a finite set of most general variants, then we are *guaranteed* that the set of most general $E \cup Ax$-unifiers of t and t' is *finite*.

At a practical level, variants are generated using narrowing (see Chapter 15 for narrowing capabilities in Maude). Narrowing with oriented equations E (with or without modulo Ax) enjoys well-known completeness results, including the generation of complete sets of unifiers and covering all rewriting sequences from instances of a term using normalized substitutions (i.e., variants). For instance, [84] showed that narrowing with E without axioms enjoyed good completeness results, and [85] showed that narrowing with E modulo axioms Ax enjoyed also good completeness results. But narrowing can be quite inefficient, generating a huge search space, and different narrowing strategies have been devised to reduce the search space while remaining complete, specially for unification purposes (see [5] for a survey on narrowing termination). The *basic narrowing strategy* of [84] provided a restriction of narrowing that, while being complete, it was terminating for specific classes of theories. However, very little was known about effective narrowing strategies in the modulo case, and some of the known anomalies ring a cautionary note, to the effect that the naive extensions of standard narrowing strategies, for example *basic narrowing modulo AC*, were incomplete [137, 36], although a recent variation of the basic narrowing strategy has been proved complete in [88]. In [71], the *folding variant narrowing strategy* is defined for the modulo case and it is proved to be complete for variants and with good termination properties, providing a finitary and complete unification algorithm for equational unification for the theories described in Section 14.3 that also satisfy the finite variant property. Moreover, it is even better than the basic narrowing strategy in the case without axioms, since it can terminate for equational theories where basic narrowing cannot (see [71]).

In Maude 2.6, variant generation and variant-based equational unification were implemented in Maude and made available in Full Maude. Instead, since Maude 2.7 variant generation and variant-based equational unification are implemented in Core Maude's C++ level for efficiency purposes and using the Ax-unification algorithm described in Section 13.4. Furthermore, the variant generation and variant-based equational unification available in Maude 2.6 were accepting only equational theories where the righthand side of the equations was a strongly irreducible term (e.g., a variable or a constant), while the current version implements the folding variant narrowing strategy in full generality. By "full generality" we mean not just any equational theory $(\Sigma, E \cup Ax)$ having the finite variant property with Ax satisfying the requirements in Section 13.3 (so that unification is *finitary*), but any confluent, coherent, and terminating modulo Ax decomposition (Σ, Ax, E), thus obtaining an incremental generation for the (in general infinite) set of $E \cup Ax$-unifiers in that case.

The key distinction, now supported for the first time in Maude in full generality, is one between *dedicated* unification algorithms for a limited set of axioms Ax and *generic* unification algorithms which can be applied to a much wider range of user-definable theories and can even deal with incremental generation of infinite sets of unifiers.

14.9 The variant unify command

Given a module ⟨*ModId*⟩, of the general form `mod` $(\Sigma, G \cup E \cup Ax, R)$ `endm` where $(\Sigma, E \cup Ax)$ satisfies the requirements of Section 14.3 and satisfies also the finite variant property, Maude provides a command for $E \cup Ax$-equational unification based on variant generation of the following two forms:

```
variant unify [ n ] in ⟨ModId⟩ :
    ⟨Term-1⟩ =? ⟨Term'-1⟩ /\ ... /\ ⟨Term-k⟩ =? ⟨Term'-k⟩ .
filtered variant unify [ n ] in ⟨ModId⟩ :
    ⟨Term-1⟩ =? ⟨Term'-1⟩ /\ ... /\ ⟨Term-k⟩ =? ⟨Term'-k⟩ .
```

where $k \geq 1$; n is an optional argument providing a bound on the number of unifiers requested, so that if the cardinality of the set of unifiers is greater than the specified bound, the unifiers beyond that bound are omitted; and, as usual, if no module is mentioned, the current module is used. The second command

Consider again the module `VARIANT-VENDING-MACHINE` introduced in Section 14.4. We can ask whether there is an $E \cup Ax$-equational unifier of two configurations, one containing at least two quarters, and another containing at least one dollar, by invoking the following command:

```
Maude> variant unify in VARIANT-VENDING-MACHINE :
        < q q X:Marking > =? < $ Y:Marking > .

Unifier 1
X:Marking --> $ %1:Marking
Y:Marking --> q q %1:Marking

Unifier 2
X:Marking --> q q #1:Marking
Y:Marking --> #1:Marking
```

It may not be obvious that this is not the minimal set of most general unifiers, but the filtered version returns only one unifier.

```
Maude> filtered variant unify in VARIANT-VENDING-MACHINE :
        < q q X:Marking > =? < $ Y:Marking > .

Unifier 1
X:Marking --> q q #1:Marking
Y:Marking --> #1:Marking
```

The first unifier is an instance of the second unifier by applying `#1:Marking --> q q` and simplifying `q q q q` into `$`. Instead, the second unifier is *not* an instance of the first unifier because the `$` in normal form disables any possibility.

Note that there are equational theories where two unifiers are comparable in both directions, i.e. unifier σ_1 is an instance of unifier σ_2 modulo the equational theory and viceversa. In such a case, Maude arbitrarily returns one of them. For instance, it is well-known that unification in the exclusive-or theory is unitary [86] and, for the following unification problem, the `variant unify` command returns 57 unifiers whereas the `filtered variant unify` command returns just one, but several could be appropriate candidates for most general unifier.

```
Maude> filtered variant unify in EXCLUSIVE-OR : X * Y =? Z * W .

Unifier 1
X --> %1:[NatSet] * %3:[NatSet]
```

```
Y --> %2:[NatSet] * %4:[NatSet]
Z --> %1:[NatSet] * %2:[NatSet]
W --> %3:[NatSet] * %4:[NatSet]
```

14.10 Variant-based unification with irreducibility constraints

Similarly to Section 14.5, we may need to perform variant-based unification but with some irreducibility conditions. The irreducibility condition is appended by using the keywords `such that T1, T2, T3, ... irreducible`.

For example, we may reconsider the unification problem of terms `< q q X:Marking >` and `< $ Y:Marking >` above but assuming that different subterms separated by commas are irreducible.

```
variant unify in VARIANT-VENDING-MACHINE : < q q X:Marking > =? < $ Y:Marking >
such that q q X:Marking, q X:Marking, X:Marking irreducible .

Unifier 1
rewrites: 0 in 0ms cpu (0ms real) (0 rewrites/second)
X:Marking --> $ %1:Marking
Y:Marking --> q q %1:Marking
```

This command clearly discards the second unifier where `X:Marking` is mapped to `q q %1:Marking`, since violates the condition that `q q X:Marking` must be irreducible under substitution.

14.11 Incremental variant unification

Similarly to the incremental generation of variants in Section 14.6, one can obtain an incremental number of unifiers for a given unification problem. Let us consider again the `NAT-VARIANT` module in Section 14.1 that does not have the finite variant property. On the one hand, it is possible to have a finite number of most general unifiers for a unification problem although the theory does not have the finite variant property. For instance, the unification problem between `s(0) + X` and `s(s(s(0)))` returns just one unifier.

```
Maude> variant unify in NAT-VARIANT : s(0) + X:Nat =? s(s(s(0))) .

Unifier 1
X:Nat --> s(s(0))

No more unifiers.
```

On the other hand, we can approximate the number of unifiers of a unification problem that we suspect does not have a finite number of most general unifiers. For instance, the unification problem between terms `X + s(0)` and `s(s(s(0)))` has only one solution $\texttt{X} \mapsto \texttt{s(s(0))}$ and we can obtain that solution by including a bound in the command, as it is also done for variant generation.

```
Maude> variant unify [1] in NAT-VARIANT : X:Nat + s(0) =? s(s(s(0))) .

Unifier 1
X:Nat --> s(s(0))
```

However, if we tried to obtain two unifiers, Maude would not stop because it would keep trying to generate a second unifier for a unification problem that has only one unifier, without knowing that it could stop. This differs from the incremental generation of variants (Section 14.6), where we can incrementally increase the bound even if the theory does not have the finite variant property and Maude will always stop with a new variant. The problem here is due to the fact that Maude needs to compute the set of variants before computing the set of unifiers; when the equational theory does not have the finite variant property, such a set of variants can be infinite and Maude is not able to complete this computation, even when the user provides a bound, because such a bound refers to the number of requested unifiers, but not to the number of variants, which in this process is just part of the internal process for computing unifiers. As in the example above, this can happen even when there are no further unifiers.

14.12 Variant unification in incomplete unification examples

Similarly to the incomplete variant generation in Section 14.7, we can have variant unification calls that cannot provide a finitary set of most general unifiers. Let us consider again the `VARIANT-UNIFICATION-ASSOC` theory of Section 14.7.

We can force unification calls, within variant unification, to an associative unification problem with an infinite set of most general unifiers, e.g., the previously described associative unification problem `N:Nat : L:NList` $=^?$ `L:NList : N;Nat` with the family of substitutions $\{$`L:NList` $\mapsto$ `N:Nat : ··· : N:Nat`$\}$:

```
Maude> variant unify in VARIANT-UNIFICATION-ASSOC :
    head(L) =? last(L) /\ prefix(L) =? tail(L) .

Warning: Unification modulo the theory of operator _:_ has encountered
an instance for which it may not be complete.

Unifier 1
L --> %1:Nat : %1:Nat : %1:Nat

Unifier 2
L --> %1:Nat : %1:Nat

No more unifiers.
Warning: Some unifiers may have been missed due to incomplete unification algorithm(s).
```

Notice that the unification problem `head(L) =? last(L) /\ prefix(L) =? tail(L)` has the same solutions as the unification problem `N : L =? L : N`.

14.13 The `variant match` command

Given a module ⟨*ModId*⟩, of the general form `mod` $(\Sigma, G \cup E \cup Ax, R)$ `endm` where $(\Sigma, E \cup Ax)$ satisfies the requirements of Section 14.3 and has also the finite variant property, Maude provides a command for $E \cup Ax$-equational matching based on variant generation of the form:

`variant match [` *n* `] in` ⟨*ModId*⟩ `:`
⟨*Term-1*⟩ `<=?` ⟨*Term'-1*⟩ `/\ ... /\` ⟨*Term-k*⟩ `<=?` ⟨*Term'-k*⟩ `.`

where $k \geq 1$; n is an optional argument providing a bound on the number of matches requested and, as usual, if no module is mentioned, the current module is used.

Variant matching works like variant unification, except that the right hand side of each matching problem is considered as a ground term, with variables treated as constants. Operationally, a slightly different algorithm to that for variant unification is used and variant generation is never applied to the right hand sides of the matching problems.

Consider again the module `VARIANT-VENDING-MACHINE` introduced in Section 14.4. We can ask whether a configuration containing at least two quarters can match a configuration containing at least one dollar, by invoking the following command:

```
Maude> variant match in VARIANT-VENDING-MACHINE :
        < q q X:Marking > <=? < $ Y:Marking > .

Matcher 1
X:Marking --> q q Y:Marking
```

Chapter 15

Narrowing

15.1 Introduction

Narrowing is a generalization of term rewriting that allows logical variables in terms (as in logic programming) and replaces pattern matching by unification in order to symbolically evaluate these terms with given rewrite rules. Narrowing is a simple, precise answer to the question:

> *Given a term t with variables $x_1, \ldots, x_n$, what are the most general instances of t that can be rewritten by the given rules?*

For example, t can be $n + m$, and the rewrite rules can be $0 + y \to y$ and $s(x) + y \to s(x + y)$. The term $n + m$ cannot be rewritten because of the variable n and finding the most general instances becomes interesting; note that if $+$ is commutative, then either n or m stop t from being rewritten.

Narrowing was originally introduced as a mechanism for solving equational unification problems [73]. It was later generalized to solve the more general problem of symbolic reachability [114]. The narrowing mechanism has a number of important applications, including automated proofs of termination [7], execution of functional-logic programming languages [77, 39, 80, 126, 103], partial evaluation [4], verification of cryptographic protocols [114], and equational unification [84], to mention just a few.

At each rewriting step one must choose which subterm of the subject term and which rule of the specification are going to be considered. Similarly, at each narrowing step one must choose which subterm of the subject term, which rule of the specification, and which instantiation on the variables of the subject term and the rule's lefthand side are going to be considered. The narrowing relation is formally defined as follows. Let $\mathcal{R} = (\Sigma, E \cup Ax, R)$ be an order-sorted rewrite theory where R is a set of *unconditional* rewrite rules specified with the `rl` keyword, E is a set of *unconditional* equations specified with the `eq` keyword, and Ax is a set of commonly occurring axioms declared in Maude as equational attributes (see Section 4.4.1) such that an $E \cup Ax$-unification procedure is available in Maude.[1] Let $CSU_{E \cup Ax}(u = u')$ provide[2]

[1] $E \cup Ax$-unification is available via the *variant*-based equational unification explained in Section 14.9. Note, however, that Maude will only perform $E \cup Ax$-unification with those equations E that have been declared with the `variant` attribute. In particular, if no equation in E has the `variant` attribute, Maude will only perform the Ax-unification explained in Section 13.4.

[2] Although Sections 13.4 and 14.9 provide a command for computing minimal sets of, respectively, unifiers modulo axioms or modulo a convergent equational theory, at present the complete sets of unifiers used by the narrowing commands are not minimal. Support for narrowing with a minimal set of unifiers is planned for a future release.

a finitary and complete set of unifiers for any pair of terms u, u' with the same top sort. The $R, E \cup Ax$-*narrowing* relation on $T_\Sigma(X)$ is defined as $t \leadsto_{\sigma,p,R,E\cup Ax} t'$ (or $\leadsto_\sigma$ when $p, R, E \cup Ax$ are understood) if there is a non-variable position p of t, a (possibly renamed) rule $l \to r$ in R, and a unifier $\sigma \in CSU_{E\cup Ax}(t|_p = l)$ such that $t' = \sigma(t[r]_p)$. We denote by $t \leadsto^+_{\sigma,R,E\cup Ax} t'$ (resp. $t \leadsto^*_{\sigma,R,E\cup Ax} t'$) the transitive (resp. reflexive-transitive) closure of the narrowing relation, where σ is obtained as the composition of the substitutions for each narrowing step in the sequence.

The difference between a rewriting step and a narrowing step is that in both cases we use a rewrite rule $l \to r$ to rewrite t at a position p in t, but narrowing *unifies* the lefthand side l and the chosen subject term $t|_p$ before actually performing the rewriting step. Also, narrowing is usually[3] restricted to non-variable positions of t, whereas rewriting does not require such a restriction.

Consider the following system module defining the addition function `_+_` on natural numbers built from `0` and `s`:

```
mod NAT-NARROWING is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars X Y : Nat .
  rl [base] : 0 + Y => Y .
  rl [ind] : s(X) + Y => s(X + Y) .
endm
```

Consider the term `X + s(0)` and the two rules `base` and `ind`. Narrowing will instantiate variable `X` with `0` and `s(X')` respectively in order to be able to apply each of these rules, i.e., the following two narrowing steps are generated:

$$\mathtt{X} + \mathtt{s(0)} \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\},\mathtt{base}} \mathtt{s(0)}$$

$$\mathtt{X} + \mathtt{s(0)} \leadsto_{\{\mathtt{X}\mapsto\mathtt{s(\#1:Nat)}\},\mathtt{ind}} \mathtt{s(\#1:Nat + s(0))}$$

Note that, for simplicity, we show only the bindings of the unifier that affect the input term. There are infinitely many narrowing derivations starting at the input expression `X + s(0)` (at each step the reduced subterm is underlined):

1. $\underline{\mathtt{X} + \mathtt{s(0)}} \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\},\mathtt{base}} \mathtt{s(0)}$
2. $\underline{\mathtt{X} + \mathtt{s(0)}} \leadsto_{\{\mathtt{X}\mapsto\mathtt{s(\#1:Nat)}\},\mathtt{ind}} \mathtt{s}(\underline{\mathtt{\#1:Nat + s(0)}})$
   $\leadsto_{\{\mathtt{\#1:Nat}\mapsto\mathtt{0}\},\mathtt{base}} \mathtt{s(s(0))}$
3. $\underline{\mathtt{X} + \mathtt{s(0)}} \leadsto_{\{\mathtt{X}\mapsto\mathtt{s(\#1:Nat)}\},\mathtt{ind}} \mathtt{s}(\underline{\mathtt{\#1:Nat + s(0)}})$
   $\leadsto_{\{\mathtt{\#1:Nat}\mapsto\mathtt{s(\#2:Nat)}\},\mathtt{ind}} \mathtt{s(s}(\underline{\mathtt{\#2:Nat + s(0)}}))$
   $\leadsto_{\{\mathtt{\#2:Nat}\mapsto\mathtt{0}\},\mathtt{base}} \mathtt{s(s(s(0)))}$

and so on.

The following infinite narrowing derivation resulting from applying rule `ind` infinitely many times can also be shown:

$$\begin{aligned}\underline{\mathtt{X} + \mathtt{s(0)}} &\leadsto_{\{\mathtt{X}\mapsto\mathtt{s(\#1:Nat)}\},\mathtt{ind}} \mathtt{s}(\underline{\mathtt{\#1:Nat + s(0)}})\\ &\leadsto_{\{\mathtt{\#1:Nat}\mapsto\mathtt{s(\#2:Nat)}\},\mathtt{ind}} \mathtt{s(s}(\underline{\mathtt{\#2:Nat + s(0)}}))\\ &\leadsto_{\{\mathtt{\#2:Nat}\mapsto\mathtt{s(\#3:Nat)}\},\mathtt{ind}} \mathtt{s(s(s}(\underline{\mathtt{\#3:Nat + s(0)}})))\\ &\ldots\end{aligned}$$

[3] The *paramodulation* inference rule used in paramodulation-based theorem proving [117] is similar to narrowing and does not require non-variable positions.

15.2 Applications

The classical application of narrowing modulo an equational theory is to perform $E \cup Ax$-*unification* by $\vec{E}, Ax$-narrowing when the equations E are oriented into sort-decreasing, confluent, terminating, and coherent modulo Ax rules $\vec{E}$. Indeed the variant-based equational order-sorted unification algorithm of Section 14.8 is based on an E, Ax-narrowing strategy, called *folding variant narrowing* [71], that terminates when $E \cup Ax$ has the finite variant property [36], even though full E, Ax-narrowing typically does not terminate when Ax contains AC axioms [36, 71].

Instead, when the rules R are understood as *transition rules*, a completely different application of $R, E \cup Ax$-narrowing is that of *symbolic reachability analysis* [114] (see Section 13.5.2). Specifically, we will consider the case of transition systems specified by order-sorted rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup Ax, R)$ where: (i) $E \cup Ax$ has a finite and complete $E \cup Ax$-unification algorithm (see the requirements of Section 14.1), and (ii) the transition rules R are $E \cup Ax$-coherent and *topmost* (so that rewriting is always done at the top of the term). Then, narrowing is a *complete* deductive method [114] for symbolic reachability analysis, that is, for solving existential queries of the form $\exists \overline{x}\ t(\overline{x}) \rightarrow^* t'(\overline{x})$ in the sense that the formula holds for $\mathcal{R}$ iff there is a narrowing sequence $t \rightsquigarrow^*_{R, E \cup Ax} u$ such that u and t' have an $E \cup Ax$-unifier.

Furthermore, in symbolic reachability analysis we may be interested in verifying properties more general than existential questions of the form $\exists X\ t \longrightarrow^* t'$. One can also generalize the above reachability question to questions of the form $\mathcal{R}, t \models \varphi$, with φ a temporal logic formula. The papers [68, 8] show how narrowing can be used (again, both at the level of transitions with rules R and at the level of equations E) to perform *logical model checking* to verify such temporal logic formulas; this is a kind of *symbolic model checking* not in the binary decision diagram sense of "symbolic," which still remains finite-state, but in a much more general sense in which possibly infinite sets of states are finitely described by patterns with logical variables. Two distinctive features are: (i) the term t does not describe a single initial state, but a possibly infinite set of instances of t (i.e., a possibly infinite set of initial states[4]); and (ii) the set of reachable states does not have to be finite. Therefore, standard model-checking techniques may not be usable, because of a possible double infinity: in the number of initial states, and in the number of states reachable from each of those initial states.

15.3 Completeness of narrowing

Due to nontermination, narrowing behaves as a semi-decision procedure for equational unification and for reachability analysis in a wide variety of theories. However, for some particular subject terms narrowing may terminate, providing a complete set of solutions. For instance, in the Maude module `NAT-NARROWING` above, narrowing computes the solution $\{\texttt{X} \mapsto \texttt{s(Y)}\}$ for the reachability problem $\exists \texttt{X}, \texttt{Y}\ \texttt{0 + X} \rightarrow^* \texttt{s(Y)}$ and it terminates with no more solutions. Instead, for the reachability problem $\exists \texttt{X}, \texttt{Z}\ \texttt{X + s(0)} \rightarrow^* \texttt{s(s(Z))}$, narrowing computes the solution $\{\texttt{X} \mapsto \texttt{s(0)},\ \texttt{Z} \mapsto \texttt{0}\}$ but it cannot terminate because of the above infinite narrowing sequence using `ind`. Moreover, narrowing cannot prove that the reachability problem $\exists \texttt{X}\ \texttt{X + s(0)} \rightarrow^* \texttt{0}$ does not have a solution, again because of the above infinite narrowing sequence using `ind`.

Note that for any narrowing sequence $t \rightsquigarrow^*_{\sigma, R, E \cup Ax} s$, we have a corresponding rewrite sequence $\sigma(t) \rightarrow^*_{R/E \cup Ax} s'$, where, by definition, the rewrite relation $\rightarrow_{R/E \cup Ax}$ is the relation composition: $\rightarrow_{R/E \cup Ax} =_{def} (=_{E \cup Ax}); \rightarrow_R; (=_{E \cup Ax})$, i.e., the transition relation between states

[4] In general, a set of initial states may not be describable by a single pattern t, but may have a description as a finite collection of patterns $t_1, \ldots, t_n$. There is no problem in handling this more general case: the narrowing-based symbolic model checking can just take a tuple $\langle t_1, \ldots, t_n \rangle$.

in the given rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, R)$. However, only under appropriate conditions is narrowing *complete* as an equational unification algorithm [84, 85], or as a procedure to solve reachability problems [114]. That is, given a reachability problem[5] $\exists x_1, \ldots, x_k : s \rightarrow^* t$, completeness of narrowing for reachability analysis means that for each possible substitution ρ such that $\rho(s) \rightarrow^*_{R/E \cup Ax} \rho(t)$, and for all the substitutions $\sigma_1, \ldots, \sigma_n, \ldots$, provided by narrowing from s, there is an index i and a term u_i such that, if $s \rightsquigarrow_{\sigma_1, R, E \cup Ax} u_1 \cdots u_{i-1} \rightsquigarrow_{\sigma_i, R, E \cup Ax} u_i$, then there is a $E \cup Ax$-unifier τ of the equation $u_i = t$, and therefore there is a rewrite sequence $\tau(\sigma(s)) \rightarrow^*_{R/E \cup Ax} \tau(u_i)$, showing that a instance of s *reaches* an instance of t modulo the equations $E \cup Ax$. Essentially, completeness holds under the following conditions:

1. for $(R, E \cup Ax)$-normalized substitutions ρ above [114] (a stronger condition is $(R \cup E, Ax)$-normalized substitutions);
2. for *topmost rewrite theories*[6];
3. for right-linear theories and linear reachability goals[7];
4. in particular for theories that are confluent, terminating, and coherent modulo axioms Ax, as the equational theories in Maude functional modules with such properties restricted to unconditional equations.

15.4 Narrowing with simplification

As pointed out in Footnote 1, given a rewrite theory $\mathcal{R} = (\Sigma, E \cup G \cup Ax, R)$, where Ax are the axioms, and $E \cup G$ the equations orientable as rewrite rules, if only the equations E have the `variant` attribute, Maude will not perform narrowing with R modulo $E \cup G \cup Ax$, but only modulo $E \cup Ax$. This gives Maude users more flexibility, since the equations G may play a useful, auxiliary role in the applications they have in mind. In particular, a concrete, practical reason for using the decomposition $E \cup G$ is that $E \cup Ax$ may be FVP, but G may not be, so that variant $E \cup Ax$-unification is finitary, whereas variant $E \cup G \cup Ax$-unification will be infinitary and undecidable. Note that the standard case of narrowing with R modulo $E \cup G \cup Ax$ is just the special case $G = \emptyset$, which may be particularly attractive if $E \cup Ax$ is FVP. The basic intuition is that, in the most general version of narrowing supported by Maude that we now explain, we *narrow* with R modulo $E \cup Ax$, and we *normalize* with oriented equations $\vec{E} \cup \vec{G}$ modulo Ax.

Each equation in E (used for *variant* computation) must include the `variant` attribute. Note that equations in G do not contain the `variant` attribute and do not have any restriction, i.e., they can be conditional equations, with the `owise` attribute, etc. Each narrowing step of the form $t \rightsquigarrow_{\sigma,p,R,E \cup Ax} t'$ is followed by simplification using the relation $\rightarrow^!_{G \cup E, Ax}$, i.e., the combined relation $(\rightsquigarrow_{\sigma,p,R,E \cup Ax}; \rightarrow^!_{G \cup E, Ax})$ is defined as $t \rightsquigarrow_{\sigma,p,R,E \cup Ax}; \rightarrow^!_{G \cup E, Ax} t''$ iff $t \rightsquigarrow_{\sigma,p,R,E \cup Ax} t'$, $t' \rightarrow^*_{G \cup E, Ax} t''$, and t'' is $G \cup E, Ax$-irreducible. Note that this combined

[5] Equational unification can be represented in terms of a reachability problem [85] (see Section 13.5.1) and, therefore, we do not consider completeness results for equational unification here; see [5] for a survey on termination of narrowing and completeness results for equational unification and reachability problems in the free case.

[6] That is, theories where every rewrite step is performed only at the top position of the term. In such theories completeness can be simplified as follows: given $\exists X_1, \ldots, X_k \; s \rightarrow^* t$, for each ρ such that $\rho(s) \rightarrow^*_{R,Ax} \rho(t)$, and for all the substitutions $\sigma_1, \ldots, \sigma_n, \ldots$ provided by narrowing from s, there is an index i and a substitution τ such that $\rho =_{Ax} \sigma_i; \tau$.

[7] A reachability goal of the form $\exists X_1, \ldots, X_k \; s \rightarrow^* t$ is linear if s is linear and s and t do not have variables in common.

relation $(\leadsto_{\sigma,p,R,E\cup Ax}; \to^{!}_{G\cup E,Ax})$ may be incomplete, i.e., given a reachability problem of the form $t \to^{*} t'$ and a solution σ (i.e., $\sigma(t) \to^{*}_{R,G\cup E\cup Ax} \sigma(t')$), the relation $\leadsto_{\sigma,p,R,E\cup Ax}; \to^{!}_{G\cup E,Ax}$ may not be able to find a more general solution. The reason is that the equations G should also be executed by narrowing instead of rewriting to ensure completeness under appropriate conditions (see [114] and Section 15.3). However, the combination of narrowing using rules, equations, and axioms with simplification using additional equations can be quite helpful to allow built-in Maude functions such as addition or multiplication, which cannot be executed by narrowing in their predefined form. It can also be useful in other applications where specific combinations of narrowing and simplification are needed.

15.5 Theories supported for narrowing reachability

The narrowing relation is defined on top of the order-sorted variant-based unification procedure described in Section 14.8. In order to avoid clashing of algorithms and notions, we have decided that the rules used for narrowing should be identifiable, as the equations for variant unification, and clearly distinguished from standard rules in Maude. For this purpose we have defined a new attribute for equations: the keyword `narrowing`.

Let `mod` $(\Sigma, G \cup E \cup Ax, R \cup H)$ `endm` be an order-sorted system module where R is a set of rewrite rules specified with the `rl` keyword and the attribute `narrowing`, H are the remaining rewrite rules specified with the `rl` or `crl` keywords but without the attribute `narrowing`, Ax is a set of commonly occurring axioms (declared in Maude as equational attributes, see Section 4.4.1), E is a set of equations specified with the `eq` keyword and the attribute `variant` such that $(\Sigma, E \cup Ax)$ satisfies the restrictions mentioned in Section 14.8, and G are the remaining equations specified with the `eq` or `ceq` keywords but without the attribute `variant`. Furthermore, the rules R must satisfy the following extra conditions:

- R cannot contain conditional rules specified with the `crl` keyword.
- The lefthand side of a rule in R cannot be a variable. If a variable is needed, one should instead specify a new kind with an extra unary symbol grabbing the whole system's state (which would before have been matched by a single variable lefthand side). In this way, the problem of having a variable lefthand side can often be solved.
- The rules in R must be explicitly Ax-coherent (see Section 5.3). If they are not, the coherence completion algorithm of Section 21.5 should be used.

We recall again that the rules H are disregarded for narrowing modulo $E \cup Ax$, and the oriented equations G are disregarded for folding variant narrowing modulo Ax in the associated task of variant unification. However, the equations G are applied for simplification after each narrowing step (see Section 15.4), as it is performed in Maude for rewriting. Recall, again, that this combination of one narrowing step followed by equational simplification is not complete. A full treatment of rules, equations, and axioms for narrowing is outside the scope of the present implementation and is left for future work.

Furthermore, *frozen arguments* (see Section 4.4.9) are allowed for narrowing, as for rewriting; see the rewrite theory `LAST-APPEND` of page 372 below for the functions `append` and `last` written in a functional-logic style. They are given the standard meaning of not allowing any narrowing step below such frozen arguments, just as in the *context-sensitive narrowing* of [93].

Finally, we do not consider any narrowing strategy at all for solving reachability problems, i.e., all positions in a term with an admissible $R, E \cup Ax$-narrowing step are explored.

15.6 The `vu-narrow` command

Given a system module ⟨*ModId*⟩, the user can give to Maude a narrowing-based search command of the form (the prefix `vu` denotes that uses variant-based unification at each step):

`vu-narrow [` *n*`,` *m* `] in` ⟨*ModId*⟩ `:` ⟨*Term-1*⟩ ⟨*SearchArrow*⟩ ⟨*Term-2*⟩ `.`

where

- *n* is an optional argument providing a bound on the number of desired solutions;
- *m* is another optional argument stating the maximum depth of the search;
- the module ⟨*ModId*⟩ where the search takes place can be omitted;
- ⟨*Term-1*⟩ is the starting term, which typically will contain variables;
- ⟨*Term-2*⟩ is the pattern that has to be reached, which may share variables with the starting term (note that terms in the narrowing sequence will be unified with this target pattern, in contrast to the `search` command of Section 5.4.3);
- ⟨*SearchArrow*⟩ is an arrow indicating the form of the rewriting proof from ⟨*Term-1*⟩ until ⟨*Term-2*⟩:
  - `=>1` means a rewriting proof consisting of exactly one step,
  - `=>+` means a rewriting proof consisting of one or more steps,
  - `=>*` means a proof consisting of none, one, or more steps, and
  - `=>!` indicates that only narrowing sequences ending in terms describing sets of final states are allowed. Such terms describing sets of final states are called *strongly irreducible* in the sense that they cannot be further narrowed; note that this is stronger than requiring states that cannot be rewritten as in the `search` command of Section 5.4.3.

  The one step arrow `=>1` is an abbreviation of the one-or-more steps arrow `=>+` with the depth bound *m* set to `1`.

Consider, for example, the following new version of the vending machine to buy apples (`a`) or cakes (`c`) with dollars (`$`) and/or quarters (`q`). The reader can check that the only difference with the `VARIANT-VENDING-MACHINE` module in Section 14.4 is the addition of the `narrowing` attribute to the rules.

```
mod NARROWING-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : < M $ > => < M c > [narrowing] .
  rl [buy-a] : < M $ > => < M a q > [narrowing] .
  eq [change] : q q q q M = $ M [variant] .
endm
```

We can use the narrowing search command to answer the question:

Is there any combination of one or more coins that returns exactly an apple and a cake?

This can be done by searching for states that are reachable from a term `< M:Money >` and match the desired pattern at the end.

```
Maude> vu-narrow [1] in NARROWING-VENDING-MACHINE : < M:Money > =>* < a c > .

Solution 1
state: < a c #1:Money >
accumulated substitution:
M:Money --> $ q q q #1:Money
variant unifier:
#1:Money --> empty
```

The narrowing-based search returns the substitution accumulated along the narrowing sequence and the variant unifier resulting from the unification of the target term and the last expression in the narrowing sequence. Indeed both unifiers must be combined *by hand* in order to have an actual solution to the symbolic reachability problem, as shown for the previous example.

```
Solution 1
state: < a c >
accumulated substitution:
M:Money --> $ q q q
```

Note that `_ _` is an ACU symbol and that such an ACU symbol appears in the equation `change`, disallowing the basic narrowing strategy [84] to be used for equational unification and requiring the folding variant narrowing [71] to be used for equational unification.

Note that we have restricted the previous reachability problem to just one solution. Narrowing does not terminate for this reachability problem even though the above solution is indeed the *only* solution. The problem is that narrowing follows a breadth-first exploration and does not stop until the whole narrowing tree demanded by the search command is created, even though this infinite search may not yield any further solutions. The very same problem happens for the standard `search` command (see Section 5.4.3). If we increase the depth of the narrowing tree, we can experimentally observe that there are no more solutions than the one shown before.

```
Maude> vu-narrow [,5] in NARROWING-VENDING-MACHINE : < M:Money > =>* < a c > .

Solution 1
state: < a c #1:Money >
accumulated substitution:
M:Money --> $ q q q #1:Money
variant unifier:
#1:Money --> empty
```

The narrowing-based search version does not provide paths to solutions but there is a metalevel command, described in Section 17.6.11, that does provide paths.

In the previous reachability problem we can change the arrow `=>*` for reachability in zero or more steps by the arrow `=>!` for reachability in zero or more steps including only states that cannot be narrowed any more.

```
Maude> vu-narrow [,5] in NARROWING-VENDING-MACHINE : < M:Money > =>! < a c > .

No solution.
```

And surprisingly we do not find the previous solution. The reason is that the transition rules of the vending machine are not terminating for narrowing and for rewriting, so it is impossible to find a state that cannot be narrowed any more. However, if we replace the variable `M:Money` by variables of sort `Coin`, we are able to find appropriate solutions. That is, we formulate the following interesting question:

> *Is there any combination of four coins that returns an apple and a cake and is such that some extra money is left but that extra money cannot be used to buy anything else?*

The fact that some money may be left is characterized by including a variable of sort `Money` in the final state, and the fact that nothing else can be bought is characterized by using the `=>!` arrow instead of `=>*`.

```
vu-narrow [,10] in NARROWING-VENDING-MACHINE :
   < C1:Coin C2:Coin C3:Coin C4:Coin > =>! < M:Money a c > .

Solution 1
state: < a c >
accumulated substitution:
C1:Coin --> q
C2:Coin --> q
C3:Coin --> q
C4:Coin --> $
variant unifier:
M:Money --> empty

Solution 2
state: < a c >
accumulated substitution:
C1:Coin --> q
C2:Coin --> q
C3:Coin --> $
C4:Coin --> q
variant unifier:
M:Money --> empty

Solution 3
state: < a c >
accumulated substitution:
C1:Coin --> q
C2:Coin --> $
C3:Coin --> q
C4:Coin --> q
variant unifier:
M:Money --> empty

Solution 4
state: < a c >
accumulated substitution:
C1:Coin --> $
C2:Coin --> q
C3:Coin --> q
C4:Coin --> q
```

```
variant unifier:
M:Money --> empty

No more solutions.
```

Another point of interest is the occurrence of variables of the form *#n:Sort* or *%n:Sort*, which are called *fresh* and are described in Chapter 13. Unification modulo axioms usually introduces fresh variables; furthermore, narrowing introduces many fresh variables because the rule applied at each narrowing step is appropriately *renamed* so that no variable is shared by it and the current term. Indeed, the standard solution used in logic and functional-logic programming language implementations is to use a counter along each narrowing derivation to ensure that fresh variables have never been used previously in that narrowing derivation. This method is called *standardized apart* [6] and it is summarized by saying that a new version of a rule, equation or axiom is always generated before being used for unification or narrowing by renaming variables according to the counter, which is incremented afterwards. This method makes the result of a computation independent of the choice of variable names.

15.7 The `fvu-narrow` command

We have motivated in the previous sections how narrowing can be used for *symbolic reachability analysis* [114] using the `vu-narrow` command. However, narrowing builds a reachability *tree* rather than a reachability *graph*, as the `search` command of Section 5.4.3 does for rewriting with rules. We have endowed Maude with a different narrowing command, the `fvu-narrow` command, that does *fold* some branches of the narrowing reachability tree in order to create a graph, with the difference that the `search` command identifies states in the rewriting-based reachability tree that are equal modulo axioms whereas the `fvu-narrow` command identifies states in the narrowing-based reachability tree that are equal modulo axioms and equations under instantiation[8]. This is similar in nature to the *folding variant narrowing strategy* [71] of the variant generation command of Chapter 14 but using rules instead of equations. Indeed, the *logical model checking* of [68, 8] is based on earlier versions of this `fvu-narrow` command.

Given a system module ⟨*ModId*⟩, the user now can give to Maude a *folding* narrowing-based search command of the form (the prefix `f` denotes folding):

`fvu-narrow [` *n*`,` *m* `]` `in` ⟨*ModId*⟩ `:` ⟨*Term-1*⟩ ⟨*SearchArrow*⟩ ⟨*Term-2*⟩ `.`

where

- *n* is an optional argument providing a bound on the number of desired solutions;
- *m* is another optional argument stating the maximum depth of the search;
- the module ⟨*ModId*⟩ where the search takes place can be omitted;
- ⟨*Term-1*⟩ is the starting term, which typically will contain variables;
- ⟨*Term-2*⟩ is the pattern that has to be reached, which may share variables with the starting term;
- ⟨*SearchArrow*⟩ is an arrow indicating the form of the rewriting proof from ⟨*Term-1*⟩ until ⟨*Term-2*⟩:

[8] A new generated state s_2 (possibly with variables) is folded into a previously generated state s_1 (possibly with variables, different from those of s_2) if s_2 is an instance of s_1 modulo axioms and variant equations.

- `=>1` means a rewriting proof consisting of exactly one step,
- `=>+` means a rewriting proof consisting of one or more steps,
- `=>*` means a proof consisting of none, one, or more steps, and
- `=>!` indicates that only terms describing sets of canonical final states are allowed (see Section 15.6).

Consider again the `NARROWING-VENDING-MACHINE` module and the following question, both in Section 15.6.

> *Is there any combination of one or more coins that returns exactly an apple and a cake?*

We can repeat, with the same result, the `vu-narrow` command but now with the `f` at the beginning.

```
Maude> fvu-narrow [1] in NARROWING-VENDING-MACHINE : < M:Money > =>* < a c > .

Solution 1
state: < a c #1:Money >
accumulated substitution:
M:Money --> $ q q q #1:Money
variant unifier:
#1:Money --> empty
```

However, producing a reachability graph instead of a reachability tree may improve the chances of having a finite search space. If we repeat the same command but without asking for the first solution, Maude reports the unique solution but gets into a loop, since the `fvu-narrow` command cannot produce a finite search graph. However, the reason is that we are giving a logical variable of sort `Money` and any new state containing an apple or a cake cannot be an instance of the initial state. Therefore, we can use a variable of sort `Marking` instead of `Money` and the *infinite* narrowing-based reachability tree is folded into a *finite* narrowing-based reachability graph.

```
Maude> fvu-narrow in NARROWING-VENDING-MACHINE : < M:Marking > =>* < a c > .

Solution 1
state: < #1:Marking >
accumulated substitution:
M:Marking --> #1:Marking
variant unifier:
#1:Marking --> a c

No more solutions.
```

But this is not the expected solution, since it is simply instantiating variable `M:Marking` to the apple and the cake. The graph is finite, as we wanted, but there is only one state, the initial one, and every other state is folded into it, since they are obviously instances of it.

What is actually happening is that this specification of the vending machine is not well suited for taking advantage of this folding narrowing technique. Consider the following new version of the vending machine. The reader can check that the only difference with the `NARROWING-VENDING-MACHINE` module in Section 15.6 is that apples and cakes are not deposited in the bag but consumed. This is typical of a logic programming style, where computed answers, rather than normal forms, are used.

```
mod FOLDING-NARROWING-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : < M $ c > => < M > [narrowing] .
  rl [buy-a] : < M $ a > => < M q > [narrowing] .
  eq [change] : q q q q M = $ M [variant] .
endm
```

We can repeat the same command and now we get the expected result, while having a finite narrowing-based reachability graph.

```
Maude> fvu-narrow in FOLDING-NARROWING-VENDING-MACHINE : < M:Marking a c > =>* < empty > .

Solution 1
state: < #1:Marking >
accumulated substitution:
M:Marking --> $ q q q #1:Marking
variant unifier:
#1:Marking --> empty

No more solutions.
```

The graph is finite, as we wanted, since, after two narrowing steps, a state of the form `< #1:Marking >` is generated and, therefore, any other further state is folded into it. Note that a similar `vu-narrow` command does not stop. Note also that if we use a variable of sort `Money`, then the narrowing-based reachability tree is finite, since no more coins can be introduced, and both the `vu-narrow` and `fvu-narrow` commands stop.

```
Maude> vu-narrow in FOLDING-NARROWING-VENDING-MACHINE : < M:Money a c > =>* < empty > .

Solution 1
state: < #1:Money >
accumulated substitution:
M:Money --> $ q q q #1:Money
variant unifier:
#1:Money --> empty

No more solutions.
```

15.8 Narrowing with extra variables in righthand sides of rules

Although rewriting does not allow extra variables in the righthand side of rules[9], extra variables in righthand sides pose no problem for narrowing. Since rules having extra variables in the righthand side are not allowed in Maude for rewriting purposes, the attribute `nonexec` (see Section 4.5.3) must be added to such rules if one wants to use them for narrowing. The `nonexec` attribute is not taken into account by narrowing: *all* unconditional rules with the `narrowing` attribute, regardless of whether or not they include the `nonexec` attribute, are used by narrowing. Extra variables in the righthand side are a common feature of programs using narrowing as the operational evaluation mechanism, as in logic programming or functional-logic programming [80]. For further details on how to write funcional-logic programs in Maude using symbolic reachability, see [63, 64]. Let us motivate this feature with an example. Consider the following program defining the function `append` for concatenating two lists and the function `last` for returning the last element of a list:

```
mod LAST-APPEND is
  sort Success .
  op success : -> Success .
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  sort NatList .
  op nil : -> NatList .
  op _:_ : Nat NatList -> NatList .

  vars XS YS : NatList .
  vars N M X Y : Nat .

  op append : NatList NatList -> [NatList] .
  rl append(nil, YS) => YS [narrowing] .
  rl append(N : XS, YS) => N : append(XS, YS) [narrowing] .

  op last : NatList -> [Nat] .
  rl last(XS) => append(YS, N : nil) =:= XS >> N [nonexec narrowing] .

  op _>>_ : [Success] [Nat] -> [Nat] [frozen (2) strat (1 0)] .
  eq success >> X:[Nat] = X:[Nat] .

  op _=:=_ : Nat Nat -> [Success] [comm] .
  rl N =:= N => success [narrowing] .

  op _=:=_ : NatList NatList -> [Success] [comm] .
  rl XS =:= XS => success [narrowing] .
endm
```

In the rule

```
rl last(XS) => append(YS, N : nil) =:= XS >> N [nonexec narrowing] .
```

[9] Except for the case of *conditional* rules, where such extra variables may be introduced in matching or rewrite conditions (see Chapter 4). But conditional rules have been excluded from the present narrowing implementation, as explained in Section 15.5.

we have used an extra variable `N` to denote the last element of the list, forcing the rule to include the `nonexec` keyword. Furthermore, note the use of symbol `_>>_` to denote a conditional expression, instead of using the keyword `crl`, and the use of an auxiliary equality symbol `_=:=_` to denote equational unification, rather than equality. Finally, all these new symbols use kinds and a special sort `Success` in order to describe what a successful solution is; this follows a logic programming approach (as in Prolog) to success and failure, see [63, 64]. When the following expression, denoting the condition of the conditional expression,

```
append(YS, N : nil) =:= XS
```

is executed by narrowing, it will be solved by instantiating `N` in the proper way. The following reachability problem is solved by narrowing but cannot be solved by rewriting due to the extra variable in the `last` rule.[10]

```
Maude> vu-narrow [1] in LAST-APPEND : last(0 : s(0) : nil) =>! Z:Nat .

Solution 1
state: s(0)
accumulated substitution:
variant unifier:
Z:Nat --> s(0)
```

Another interesting example of narrowing with extra variables is the function `member`:

```
mod MEMBERSHIP is
  protecting LAST-APPEND .
  vars XS YS ZS : NatList .
  vars N M X Y : Nat .
  op member : Nat NatList -> [Success] .
  rl member(N, XS) => append(YS, N : ZS) =:= XS [nonexec narrowing] .
endm
```

The following reachability problem is solved by narrowing but cannot be solved by rewriting due to the extra variable in the rule defining the `member` function.

```
Maude> vu-narrow [1] in MEMBERSHIP : member(s(0), 0 : s(0) : nil) =>! success .

Solution 1
state: success
accumulated substitution:
variant unifier:
```

But the interesting application is to enumerate all the elements of a list by computing different substitutions, as in logic programming.

```
Maude> vu-narrow [,5] in MEMBERSHIP : member(N, 0 : s(0) : nil) =>! success .

Solution 1
state: success
accumulated substitution:
N --> 0
variant unifier:
```

[10] Of course, one could specify the `last` function in a way not requiring the `nonexec` rule, for example by the equations `last(nil) = error`, `last(N : nil) = N`, and `last(N : (M : XS)) = last (M : XS)`. The point here is to illustrate with a simple example the use of narrowing with rules having extra variables in their righthand sides, which is typical of logic programming and functional-logic programming.

```
Solution 2
state: success
accumulated substitution:
N --> s(0)
variant unifier:

No more solutions.
```

Note that we have to restrict the depth of the narrowing tree to five because there exists an infinite number of narrowing sequences from the term `member(N:Nat,0 : s(0): nil)` even if only two solutions exist.

Chapter 16

SMT Solving

Given a Σ-algebra A and a quantifier-free (QF) formula φ with variables Y, we say that φ is *satisfiable* (resp. *valid*) in A iff there is an assignment map $a : Y \to A$ such that $A, a \models \varphi$ (resp. for all maps $a : Y \to A$, $A, a \models \varphi$). Note that φ is valid in A iff $\neg\varphi$ is *unsatisfiable* in A. We say that *satisfiability* in A is *decidable* iff we have an algorithm such that given any QF Σ-formula φ it can answer whether or not φ is satisfiable in A. Of course, this also means that it can answer whether or not φ is *valid* in A.

Assuming that all sorts in Σ are nonempty, i.e., $T_{\Sigma,s} \neq \emptyset$ for each sort s, we can view the equational unification methods for theories (Σ, G) explained in Chapters 13 and 14, where G can be either a set of axioms Ax, or a set of FVP equations $E \cup Ax$, as providing a decision procedure for satisfiability in the initial algebra $T_{\Sigma/G}$ for any *positive* (i.e., having no negations) QF Σ-formula φ. This is so because any such positive φ can be put in disjunctive normal form as $\varphi \equiv \bigvee_{i \in I} \bigwedge Q_i$, with each $\bigwedge Q_i$ the conjunction of a set of equations Q_i. Then φ is satisfiable in $T_{\Sigma/G}$ iff there exists a $j \in I$ such that $\bigwedge Q_j$ has at least one G-unifier, which we can decide in Maude by asking for the first unifier of the system of equations $\bigwedge Q_j$.

In this chapter we broaden the picture about satisfiability begun in Chapters 13 and 14 by explaining how satisfiability of QF formulas for several concrete Σ-algebras of interest is directly supported in Core Maude by special modules and an interface of commands that makes accessible to Maude users part of the functionality of the CVC4 and Yices2 SMT solvers. The reader should note that a given Maude binary can only be connected to one of the two SMT solvers, since the choice between Yices2 and CVC4 is made when building Maude. The binaries distributed in the official website are linked with Yices2, but instructions on how to compile Maude with CVC4 are available in the compilation guide that comes with the Maude sources. In Section 16.6 we also briefly explain how satisfiability in initial algebras of the form $T_{\Sigma/E \cup Ax}$, where $(\Sigma, E \cup Ax)$ is FVP and satisfies one additional requirement, is also supported by algorithms written in Maude that use the variant unification algorithm from Chapter 14 as a subroutine.

The most basic type of satisfiability is that of formulas in *propositional logic*, that is, of Boolean expressions, in the two-element Boolean algebra $\mathbb{B}$. This is called the *SAT* problem, and decision procedures solving it are called *SAT solvers*. By definition, a Boolean expression β with propositional variables X is *satisfiable* in $\mathbb{B}$ iff there is an assignment $a : X \to \mathbb{B}$ of Boolean values for the propositional variables X of β such that β is true under the assignment a.

But note that such a *propositional* SAT-solving problem for β in $\mathbb{B}$ is equivalent to the *first-order* satisfiability in the initial algebra $\mathbb{B}$ of the equational theory of the Booleans—as specified,

for example, in the `BOOL-OPS` functional module of Chapter 7.1—of the equation $\beta = \texttt{true}$. In fact, Maude provides *three* different methods for solving the SAT problem: (1) the `BOOL-OPS` module can decide if β is satisfiable in $\mathbb{B}$: it is so iff its canonical form in `BOOL-OPS` is different from *false*;[1] (2) the LTL SAT solver described in Section 12.4, since propositional formulas are a special case of LTL formulas; and (3) the connections to the CVC4 and Yices SAT solvers explained in this chapter, which is of course the most efficient method among those three.

Satisfiability modulo theories (SMT) solvers provide satisfiability procedures both for SAT-solving and for various first-order theories T of interest. The theories discussed in this chapter are all theories T associated to specific data types of interest in the following, precise sense: given a Σ-algebra A its first-order theory $th(A)$ is the theory $th(A) = (\Sigma, \Gamma(A))$, where $\Gamma(A) = \{\varphi \mid A \models \varphi\}$. Specifically, we will consider satisfiability procedures for QF formulas in theories of the form $th(A)$ of A: (i) Presburger arithmetic on the integers $(\mathbb{Z}, 0, 1, +, >, \geq)$, (ii) linear arithmetic on the rationals $(\mathbb{Q}, Q, +, >, \geq)$, where Q is the (infinite) *set* of rational numbers interpreted by themselves as *constants* of the algebra $\mathbb{Q}$;[2] and (iii) linear arithmetic for both the integers and the rationals, corresponding, essentially, to a subsort inclusion $\mathbb{Z} \subset \mathbb{Q}$. Therefore, the intended model in this last case is in essence the linear arithmetic fragment of the initial algebra of the theory `RAT` of rationals described in Section 7.6, which contains the sort `Int` as a subsort. SMT solvers are called this way because they exploit the Boolean structure of QF first-order formulas to perform SAT solving at the Boolean level in order to achieve more efficient versions of satisfiability procedures for the theories T supported by the solver, using the so-called DPLL(T) approach [116].

To make SMT solving for the above-described theories available to Maude users, we have integrated within Maude, and have provided access to some of the decision procedures and functionality of, two state-of-the-art SMT solvers, namely, CVC4[3] and Yices2[4]. Maude's connection to both CVC4 and Yices2 relies on the standard SMT-LIB notation[5] accepted by both SMT solvers and allows for more connections to other SMT solvers to be added in the future. As already mentioned, the specific theories currently supported in Maude are: (i) SAT-solving for the Booleans, (ii) Presburger arithmetic for the integers, (iii) linear arithmetic for the rationals; and (iv) linear arithmetic for both the integers and the rationals. We describe below the corresponding Maude modules allowing definition of QF formulas in those four theories.

16.1 Boolean formulas

The SMT interface is accessed by loading the file `smt.maude`. The SMT-LIB `Core` theory is represented by the following signature.

```
fmod BOOLEAN is
  sort Boolean .
  op true : -> Boolean [special (id-hook SMT_Symbol (true))] .
```

[1] See [128] for several alternative equational specifications besides `BOOL-OPS`, as well as a detailed explanation of how satisfying assignments for satisfiable expressions can be obtained in an equational manner.

[2] This theory is the *same* as that of linear arithmetic with rational coefficients for the real numbers. That is, for the algebra $(\mathbb{R}, Q, +, >, \geq)$. The key observation is that any system of linear equations with rational coefficients has a solution in the real numbers if and only if it has a solution in the rational numbers. The real numbers are relevant only in the case of non-linear equations not supported here. Therefore, in what follows we will not make any distinction between these two identical linear arithmetic theories and will just talk about "linear arithmetic on the rationals." Likewise, the theory's main sort can be called either `Rat` or `Real`, since this makes no difference. Actually, we will use `Real` to avoid confusion with the already existing sort `Rat` in the `RAT` module.

[3] Available at `https://cvc4.github.io`.

[4] Available at `https://yices.csl.sri.com`.

[5] See `http://www.smtlib.org/`

```
  op false : -> Boolean [special (id-hook SMT_Symbol (false))] .

  op not_ : Boolean -> Boolean
     [prec 53 special (id-hook SMT_Symbol (not))] .
  op _and_ : Boolean Boolean -> Boolean
     [gather (E e) prec 55 special (id-hook SMT_Symbol (and))] .
  op _xor_ : Boolean Boolean -> Boolean
     [gather (E e) prec 57 special (id-hook SMT_Symbol (xor))] .
  op _or_ : Boolean Boolean -> Boolean
     [gather (E e) prec 59 special (id-hook SMT_Symbol (or))] .
  op _implies_ : Boolean Boolean -> Boolean
     [gather (e E) prec 61 special (id-hook SMT_Symbol (implies))] .

  op _===_ : Boolean Boolean -> Boolean
     [gather (e E) prec 51 special (id-hook SMT_Symbol (===))] .
  op _=/==_ : Boolean Boolean -> Boolean
     [gather (e E) prec 51 special (id-hook SMT_Symbol (=/==))] .
  op _?_:_ : Boolean Boolean Boolean -> Boolean
     [gather (e e e) prec 71 special (id-hook SMT_Symbol (ite))] .
endfm
```

Here a different sort `Boolean` and constants `true` and `false` are used for true and false to avoid immediately ambiguity with Maude's `Bool` sort and its constants `true` and `false`. All of the operators are inert at the Maude level, i.e., they have no built-in semantics in Maude. The `_?_:_` operator is the if-then-else operation and `_===_` and `_=/==_` are used to represent equality and inequality, again to minimize syntactic ambiguity with Maude's standard symbols.

16.2 Formulas using integer linear arithmetic

The SMT-LIB `Ints` theory is represented by the following signature.

```
fmod INTEGER is
  protecting BOOLEAN .
  sort Integer .
  op <Integers> : -> Integer [special (id-hook SMT_NumberSymbol (integers))] .

  op -_ : Integer -> Integer
     [special (id-hook SMT_Symbol (-))] .
  op _+_ : Integer Integer -> Integer
     [gather (E e) prec 33 special (id-hook SMT_Symbol (+))] .
  op _*_ : Integer Integer -> Integer
     [gather (E e) prec 31 special (id-hook SMT_Symbol (*))] .
  op _-_ : Integer Integer -> Integer
     [gather (E e) prec 33 special (id-hook SMT_Symbol (-))] .
  op _div_ : Integer Integer -> Integer
     [gather (E e) prec 31 special (id-hook SMT_Symbol (div))] .
  op _mod_ : Integer Integer -> Integer
     [gather (E e) prec 31 special (id-hook SMT_Symbol (mod))] .

  op _<_ : Integer Integer -> Boolean
     [prec 37 special (id-hook SMT_Symbol (<))] .
  op _<=_ : Integer Integer -> Boolean
     [prec 37 special (id-hook SMT_Symbol (<=))] .
  op _>_ : Integer Integer -> Boolean
```

```
      [prec 37 special (id-hook SMT_Symbol (>))] .
   op _>=_ : Integer Integer -> Boolean
      [prec 37 special (id-hook SMT_Symbol (>=))] .

   op _===_ : Integer Integer -> Boolean
      [gather (e E) prec 51 special (id-hook SMT_Symbol (===))] .
   op _=/==_ : Integer Integer -> Boolean
      [gather (e E) prec 51 special (id-hook SMT_Symbol (=/==))] .
   op _?_:_ : Boolean Integer Integer -> Integer
      [gather (e e e) prec 71 special (id-hook SMT_Symbol (ite))] .

   op _divisible_ : Integer Integer -> Boolean
      [prec 51 special (id-hook SMT_Symbol (divisible))] .
endfm
```

Here a different sort `Integer` is given for the integer constants $\ldots, -2, -1, 0, 1, 2, \ldots$ to avoid ambiguity with the integer constants of Maude's `Int` sort. Note that `_divisible_` has the opposite argument order to Maude's `_divides_` operator on `Int` but it follows the SMT-LIB convention.

16.3 Formulas using rational linear arithmetic

The SMT-LIB `Reals` theory is represented by the following signature:

```
fmod REAL is
  protecting BOOLEAN .
  sort Real .
  op <Reals> : -> Real [special (id-hook SMT_NumberSymbol (reals))] .

  op -_ : Real -> Real [special (id-hook SMT_Symbol (-))] .
  op _+_ : Real Real -> Real
     [gather (E e) prec 33 special (id-hook SMT_Symbol (+))] .
  op _*_ : Real Real -> Real
     [gather (E e) prec 31 special (id-hook SMT_Symbol (*))] .
  op _-_ : Real Real -> Real
     [gather (E e) prec 33 special (id-hook SMT_Symbol (-))] .
  op _/_ : Real Real -> Real
     [gather (E e) prec 31 special (id-hook SMT_Symbol (/))] .

  op _<_ : Real Real -> Boolean
     [prec 37 special (id-hook SMT_Symbol (<))] .
  op _<=_ : Real Real -> Boolean
     [prec 37 special (id-hook SMT_Symbol (<=))] .
  op _>_ : Real Real -> Boolean
     [prec 37 special (id-hook SMT_Symbol (>))] .
  op _>=_ : Real Real -> Boolean
     [prec 37 special (id-hook SMT_Symbol (>=))] .

  op _===_ : Real Real -> Boolean
     [gather (e E) prec 51 special (id-hook SMT_Symbol (===))] .
  op _=/==_ : Real Real -> Boolean
     [gather (e E) prec 51 special (id-hook SMT_Symbol (=/==))] .
  op _?_:_ : Boolean Real Real -> Real
     [gather (e e e) prec 71 special (id-hook SMT_Symbol (ite))] .
```

```
endfm
```

Here a different sort `Real` is given for the rational constants that look like $-1/1$, $-3/100$, $0/1$, $7/3$, ... to avoid ambiguity with the rational constants of Maude's `Rat` sort. The / symbol is mandatory and servers to distinguish the reals from the integers (which are completely different types in SMT-LIB). Again all of the operators are inert, although currently, rational constants are canonicalized (this may change).

16.4 Formulas using rational and integer linear arithmetic

Finally, the SMT-LIB `Reals-Ints` theory is represented by the following signature:

```
fmod REAL-INTEGER is
  protecting INTEGER .
  protecting REAL .

  op toReal : Integer -> Real [special (id-hook SMT_Symbol (toReal))] .
  op toInteger : Real -> Integer [special (id-hook SMT_Symbol (toInteger))] .
  op isInteger : Real -> Boolean [special (id-hook SMT_Symbol (isInteger))] .
endfm
```

16.5 Satisfiability of formulas

The SMT solver is invoked internally using the `check` command. The syntax of this command is:

`check` [`in` ⟨*ModId*⟩ `:`] ⟨*Term*⟩ .

where *Term* must only contain the constants and operators from the appropriate signatures above, together with variables from the three sorts, `Boolean`, `Integer` and `Real` when these are available. Adding any super or subsorts to these three sorts (or subsort relations between them) will cause inconsistency issues, as will adding any additional operators.

Maude responds with whatever the SMT solver returns, typically `sat` for satisfiable or `unsat` for unsatisfiable. Note that the current implementation is quite brittle. There is only a cursory attempt to check that the query to the SMT solver makes sense and no attempt to catch exceptions. Consider the following example.

```
fmod TEST-RI is
  protecting REAL-INTEGER .
  vars W X Y Z : Boolean .
  vars I J K L : Integer .
  vars P Q R S : Real .
endfm

Maude> check toInteger(R) + toInteger(P) === toInteger(R + P) .
Result from sat solver is: sat

Maude> check not(toInteger(R) + toInteger(P) === toInteger(R + P)) .
Result from sat solver is: sat

Maude> check -2 < I and -2 * I > -1 and I =/== -1 .
Result from sat solver is: sat
```

```
Maude> check -2 < I and -2 * I > -1 and I =/== -1 and I - I =/== I .
Result from sat solver is: unsat
```

16.6 A brief introduction to variant satisfiability

A current limitation of SMT solvers is that they only provide *domain-specific* satisfiability procedures for a fixed family $T_1, \ldots, T_n$ of theories and their so-called Nelson-Oppen combinations [115]. To verify programs in a conventional programming language with a fixed set of data types this is quite reasonable. But in a language like Maude, where order-sorted data types are *user-definable* as the initial models of functional modules, it would be highly desirable to have *theory-generic* decision procedures for an infinite class of *user-definable* theories of the form $th(T_{\Sigma/E \cup Ax})$, that is, for the theory of *all inductive theorems* true in the initial algebra $T_{\Sigma/E \cup Ax}$, where $(\Sigma, E \cup Ax)$ belongs to an infinite class of theories specifiable as functional modules in Maude. Specifically, such a class consists of equational theories that: (i) are FVP and (ii) protect a constructor subspecification $(\Omega, E_\Omega \cup Ax_\Omega)$ satisfying the *OS-compactness* property [109]. The OS-compactness requirement is quite mild in practice. For example, all constructor theories where the constructors are free modulo axioms, i.e., of the form (Ω, Ax_Ω), and where any associativity axiom in Ax_Ω has a corresponding commutative axiom also in Ax_Ω is OS-compact, and so are as well various parameterized theories of interest for lists, compact lists, multisets, sets, and hereditarily finite sets [109]. The theory-generic satisfiability algorithm for theories in this class is called *variant satisfiability* [109]. Detailed algorithms for variant satisfiability and its associated auxiliary functions (which invoke variant unification as a subroutine) as well as a Maude prototype tool are described in [131]. This is a quite active area of research with many applications. In the near future two important research objectives are to: (i) improve and document the current variant satisfiability prototype; and (ii) combine within Maude: (a) domain-specific decision procedures from state-of-the-art SMT solvers, (b) order-sorted congruence closure modulo axioms [108], and (c) variant satisfiability.

Chapter 17

Reflection, Metalevel Computation, and Internal Strategies

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic which can be faithfully interpreted in itself. Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective [32, 25, 33, 34]. This makes the metatheory of rewriting logic accessible to the user in a clear and principled way. However, since a naive implementation of reflection can be computationally expensive, a good implementation must provide efficient ways of performing reflective computations. This chapter explains how this is achieved in Maude through its predefined `META-LEVEL` module, that can be found in the `prelude.maude` file.

17.1 Reflection and metalevel computation

Rewriting logic is reflective in a precise mathematical way, namely, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{R}}$, any terms t, t' in $\mathcal{R}$ as terms $\overline{t}, \overline{t'}$, and any pair $(\mathcal{R}, t)$ as a term $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, in such a way that we have the following equivalence

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

Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection:

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

In this chain of equivalences we say that the first rewriting computation takes place at level 0, the second at level 1, and so on. In a naive implementation, each step up the reflective tower comes at considerable computational cost, because simulating a single step of rewriting at one level involves many rewriting steps one level up. It is therefore important to have systematic ways of lowering the levels of reflective computations as much as possible, so that a rewriting subcomputation happens at a higher level in the tower only when this is strictly necessary.

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`. This module includes the modules `META-VIEW`, `META-MODULE`, `META-STRATEGY`, and `META-TERM`. As an overview,

- in the module `META-TERM`, Maude terms are metarepresented as elements of a data type `Term` of terms;
- in the module `META-STRATEGY`, the Maude strategy language is metarepresented as terms in a data type `Strategy` of strategy expressions;
- in the module `META-MODULE`, Maude modules are metarepresented as terms in a data type `Module` of modules;
- in the module `META-VIEW`, Maude views are metarepresented as terms in a data type `View` of views; and
- in the module `META-LEVEL`,
  - operations `upModule`, `upTerm`, `downTerm`, and others allow moving between reflection levels;
  - the process of reducing a term to canonical form using Maude's `reduce` command is metarepresented by a built-in function `metaReduce`;
  - the processes of rewriting a term in a system module using Maude's `rewrite` and `frewrite` commands are metarepresented by built-in functions `metaRewrite` and `metaFrewrite`;
  - the process of applying (without extension) a rule of a system module at the top of a term is metarepresented by a built-in function `metaApply`;
  - the process of applying (with extension) a rule of a system module at any position of a term is metarepresented by a built-in function `metaXapply`;
  - the process of matching (without extension) two terms at the top is reified by a built-in function `metaMatch`;
  - the process of matching (with extension) a pattern to any subterm of a term is reified by a built-in function `metaXmatch`;
  - the process of searching for a term satisfying some conditions starting in an initial term is reified by built-in functions `metaSearch` and `metaSearchPath`;
  - the processes of rewriting a term using Maude's `srewrite` and `dsrewrite` commands are metarepresented by built-in functions `metaSrewrite` and `metaDsrewrite`; and
  - parsing and pretty-printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also metarepresented by corresponding built-in functions.

The functions `metaReduce`, `metaApply`, `metaXapply`, `metaRewrite`, `metaFrewrite`, `metaMatch`, `metaXmatch`, `metaSrewrite`, and `metaDsrewrite` are called *descent functions*, since they allow us to descend levels in the reflective tower. The paper [28] provides a formal definition of the notion of *descent function*, and a detailed explanation of how they can be used to achieve a systematic, conservative way of lowering the levels of reflective computations.

The importation graph in Figure 17.1 shows the relationships between all the modules in the metalevel. The modules `NAT-LIST` and `QID-LIST` provide lists of natural numbers and quoted identifiers, respectively (see Section 7.13.1), and the module `QID-SET` provides sets of quoted identifiers (see Section 7.13.2). Notice that `QID-SET` is imported (in protecting mode) with renaming

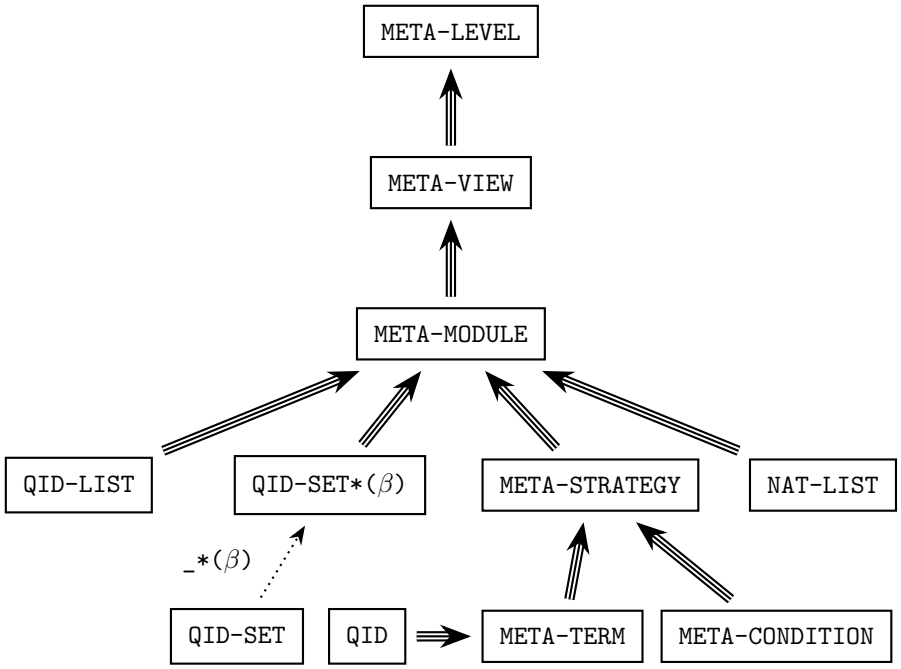


Figure 17.1: Importation graph of metalevel modules

```
(op empty to none, op _,_ to _;_ [prec 43])
```

abbreviated to β in the figure.

17.2 The META-TERM module

17.2.1 Metarepresenting sorts and kinds

In the META-TERM module, sorts and kinds are metarepresented as data in specific subsorts of the sort Qid of quoted identifiers.

A term of sort Sort is any quoted identifier not containing the following characters: ‘:’, ‘.’, ‘[’, and ‘]’. Moreover, the characters ‘{’, ‘}’, and ‘,’ can only appear in structured sort names (see Section 3.3). For example, 'Bool, 'NzNat, a‘{X‘}, a‘{X‘,Y‘}, a‘{b‘,c‘{d‘}‘}‘{e‘}, and a‘{‘(‘} are terms of sort Sort.

An element of sort Kind is a quoted identifier of the form ''[*SortList*‘] where *SortList* is a single identifier formed by a list of unquoted elements of sort Sort separated by backquoted commas. For example, ''[Bool‘] and ''[NzNat‘,Zero‘,Nat‘] are valid elements of the sort Kind. Note the use of backquotes to force them to be single identifiers.

Since commas and square brackets are used to metarepresent kinds, these characters are forbidden in sort names, in order to avoid undesirable ambiguities. Periods and colons are also forbidden, due to the metarepresentation of constants and variables, as explained in the next section.

Since operator declarations can use both sorts and kinds, we denote by Type the union of Sort and Kind.

```
sorts Sort Kind Type .
subsorts Sort Kind < Type < Qid.
op <Qids> : -> Sort [special (...)] .
op <Qids> : -> Kind [special (...)] .
```

Remember from the introduction of Chapter 7 that `<Qids>` is a special operator declaration used to represent sets of constants that are not algebraically constructed, but are instead associated with appropriate C++ code by "hooks" which are specified following the `special` attribute; see the functional module `META-TERM` in file `prelude.maude` for the details omitted here.

17.2.2 Metarepresenting terms

In the module `META-TERM`, terms are metarepresented as elements of the data type `Term` of terms. The base cases in the metarepresentation of terms are given by subsorts `Constant` and `Variable` of the sort `Qid`.

```
sorts Constant Variable Term .
subsorts Constant Variable < Qid Term .
op <Qids> : -> Constant [special (...)] .
op <Qids> : -> Variable [special (...)] .
```

Constants are quoted identifiers that contain the constant's name and its type separated by a '`.`', e.g., `'0.Nat`. Similarly, variables contain their name and type separated by a '`:`', e.g., `'N:Nat`. Appropriate selectors then extract their names and types.

```
op getName : Constant -> Qid .
op getName : Variable -> Qid .
op getType : Constant -> Type .
op getType : Variable -> Type .
```

Since '`.`' and '`:`' are not allowed in sort names (see Section 3.3), the name and type of a constant or variable can be calculated easily. Note that there is no restriction in operator or in variable names, and thus the scanning for '`.`' or '`:`' is done from right to left in the identifier. That is,

```
getName(':-D:Smile) = ':-D
getType(':-.|.`[Smile`]) = '`[Smile`]
```

A term different from a constant or a variable is constructed in the usual way, by applying an operator symbol to a nonempty list of terms.

```
sorts NeTermList TermList .
subsorts Term < NeTermList < TermList .
op _,_ : TermList TermList -> TermList
    [ctor assoc id: empty gather (e E) prec 121] .
op _,_ : NeTermList TermList -> NeTermList [ctor ditto] .
op _,_ : TermList NeTermList -> NeTermList [ctor ditto] .
op _[_] : Qid NeTermList -> Term [ctor] .
```

The actual sort infrastructure provided by the module `META-TERM` is a bit more complex, because there are also subsorts and operators for the metarepresentation of ground terms and the corresponding lists of ground terms that we do not describe here (see the file `prelude.maude` for details).

Since terms in the module `META-TERM` can be metarepresented just as terms in any other module, the metarepresentation of terms can be iterated.

For example, the term `c q M:Marking` in the module `VENDING-MACHINE` in Section 5.1 is metarepresented by

```
'__['c.Item, '__['q.Coin, 'M:Marking]]
```

and meta-metarepresented by

```
'_`[_`]['']__.Qid,
       '_`,_['']c.Item.Constant,
            '_`[_`]['']__.Qid,
                  '_`,_['']q.Coin.Constant,
                       ''M:Marking.Variable]]]]
```

Note that the metarepresentation of a natural number such as, e.g., 42 is `'s_^42['0.Zero]` instead of `'42.NzNat`, since, as explained in Section 7.2, 42 is just syntactic sugar for `s_^42(0)`.

17.3 The META-STRATEGY module: Metarepresenting the strategy language

All components of the strategy language described in Section 10, including its modules and views, can be manipulated at the metalevel. The `META-STRATEGY` module metarepresents all the strategy combinators as terms of the sort `Strategy`, with two subsorts, `RuleApplication` for rule applications, and `CallStrategy` for strategy calls.

```
ops fail idle all : -> Strategy [ctor] .
op _[_]{_} : Qid Substitution StrategyList -> RuleApplication [ctor prec 21] .
op top : RuleApplication -> Strategy [ctor] .
op match_s.t._ : Term EqCondition -> Strategy [ctor prec 21] .
op xmatch_s.t._ : Term EqCondition -> Strategy [ctor prec 21] .
op amatch_s.t._ : Term EqCondition -> Strategy [ctor prec 21] .
op _|_ : Strategy Strategy -> Strategy
                            [ctor assoc comm id: fail prec 41 gather (e E)] .
op _;_ : Strategy Strategy -> Strategy [ctor assoc id: idle prec 39 gather (e E)] .
op _+ : Strategy -> Strategy [ctor] .
op _?_:_ : Strategy Strategy Strategy -> Strategy [ctor prec 55] .
op matchrew_s.t._by_ : Term EqCondition VarStratList -> Strategy [ctor] .
op xmatchrew_s.t._by_ : Term EqCondition VarStratList -> Strategy [ctor] .
op amatchrew_s.t._by_ : Term EqCondition VarStratList -> Strategy [ctor] .
op _[[_]] : Qid TermList -> CallStrategy [ctor prec 21] .
op one : Strategy -> Strategy [ctor] .
```

The syntax is similar to that at the object level, including the strategy list for rule applications, and the substrategies lists in the `matchrew` operator.

```
op empty : -> StrategyList [ctor] .
op _,_ : StrategyList StrategyList -> StrategyList [ctor assoc id: empty] .
op _using_ : Variable Strategy -> UsingPair [ctor prec 21] .
op _,_ : UsingPairSet UsingPairSet -> UsingPairSet [ctor assoc comm prec 61] .
eq U:UsingPair, U:UsingPair = U:UsingPair .
```

Conditions are described in the module `META-CONDITION` included by `META-STRATEGY`.

```
fmod META-CONDITION is
  protecting META-TERM .

  sorts EqCondition Condition .
```

```
  subsort EqCondition < Condition .
  op nil : -> EqCondition [ctor] .
  op _=_ : Term Term -> EqCondition [ctor prec 71] .
  op _:_ : Term Sort -> EqCondition [ctor prec 71] .
  op _:=_ : Term Term -> EqCondition [ctor prec 71] .
  op _=>_ : Term Term -> Condition [ctor prec 71] .
  op _/\_ : EqCondition EqCondition -> EqCondition
      [ctor assoc id: nil prec 73] .
  op _/\_ : Condition Condition -> Condition
      [ctor assoc id: nil prec 73] .
endfm
```

The derived strategy combinators, explained in the last part of Section 10.1, are also defined as constructors. For efficiency purposes they are encoded internally in a special form, different from the encoding of their equivalent expressions.

```
op _or-else_ : Strategy Strategy -> Strategy [ctor assoc prec 43 gather (e E)] .
op _* : Strategy -> Strategy [ctor] .
op _! : Strategy -> Strategy [ctor] .
op not : Strategy -> Strategy [ctor] .
op test : Strategy -> Strategy [ctor] .
op try : Strategy -> Strategy [ctor] .
```

For example, the metarepresentation of the strategy `one(move * ; amatch (0)[nil])` for the `HANOI` example in Section 10 is

```
one('move[none]{empty} * ; amatch ''(_`)`[_`]['0.Zero, 'nil.NatList] s.t. nil)
```

17.4 The `META-MODULE` module: Metarepresenting modules

In the module `META-MODULE`, which imports `META-TERM`, functional, system and strategy modules, as well as functional, system and strategy theories, 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 in Section 17.2.2;

2. in the metarepresentation of modules and theories we follow a fixed order in introducing the different kinds of declarations for sorts, subsort relations, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way;

3. there is no need for variable declarations—in fact, there is no syntax for metarepresenting them—and

4. sets of identifiers—used in declarations of sorts—are metarepresented as sets of quoted identifiers built with an associative and commutative operator `_;_`.

The syntax for the top-level operators metarepresenting modules and theories is as follows, where `Header` means just an identifier in the case of non-parameterized modules or an identifier together with a list of parameter declarations in the case of a parameterized module.

```
sorts FModule SModule StratModule FTheory STheory StratTheory Module .
subsorts FModule < SModule < Module .
subsorts FTheory < STheory < Module .
subsorts StratModule StratTheory < Module .
sort Header .
subsort Qid < Header .
op _{_} : Qid ParameterDeclList -> Header [ctor] .
op fmod_is_sorts_.____endfm : Header ImportList SortSet
     SubsortDeclSet OpDeclSet MembAxSet EquationSet -> FModule
     [ctor gather (& & & & & &)] .
op mod_is_sorts_._____endm : Header ImportList SortSet
     SubsortDeclSet OpDeclSet MembAxSet EquationSet RuleSet
     -> SModule [ctor gather (& & & & & & &)] .
op smod_is_sorts_._______endsm : Header ImportList SortSet
     SubsortDeclSet OpDeclSet MembAxSet EquationSet RuleSet
     StratDeclSet StratDefSet -> StratModule
     [ctor gather (& & & & & & & & &)] .
op fth_is_sorts_.____endfth : Qid ImportList SortSet SubsortDeclSet
     OpDeclSet MembAxSet EquationSet -> FTheory
     [ctor gather (& & & & & &)] .
op th_is_sorts_._____endth : Qid ImportList SortSet SubsortDeclSet
     OpDeclSet MembAxSet EquationSet RuleSet -> STheory
     [ctor gather (& & & & & & &)] .
op sth_is_sorts_._______endsth : Qid ImportList SortSet
     SubsortDeclSet OpDeclSet MembAxSet EquationSet RuleSet
     StratDeclSet StratDefSet -> StratTheory
     [ctor gather (& & & & & & & & &)] .
```

Appropriate selectors then extract from the metarepresentation of modules the metarepresentations of their names, imported submodules, and declared sorts, subsorts, operators, memberships, equations, and rules.

```
op getName : Module -> Qid .
op getImports : Module -> ImportList .
op getSorts : Module -> SortSet .
op getSubsorts : Module -> SubsortDeclSet .
op getOps : Module -> OpDeclSet .
op getMbs : Module -> MembAxSet .
op getEqs : Module -> EquationSet .
op getRls : Module -> RuleSet .
op getStrats : Module -> StratDeclSet .
op getSds : Module -> StratDefSet .
```

Without going into all the syntactic details, we show only the operators used to metarepresent sets of sorts and kinds, equations, rules and strategies. The complete syntax used for metarepresenting modules can be found in the module `META-MODULE` in the file `prelude.maude`. Conditions are defined in the module `META-CONDITION` shown in Section 17.3.

```
sorts EmptyTypeSet NeSortSet NeKindSet
     NeTypeSet SortSet KindSet TypeSet .
subsort EmptyTypeSet < SortSet KindSet < TypeSet < QidSet .
subsort Sort < NeSortSet < SortSet .
subsort Kind < NeKindSet < KindSet .
subsort Type NeSortSet NeKindSet < NeTypeSet < TypeSet NeQidSet .
op none : -> EmptyTypeSet [ctor] .
```

```
op _;_ : TypeSet TypeSet -> TypeSet
     [ctor assoc comm id: none prec 43] .
op _;_ : SortSet SortSet -> SortSet [ctor ditto] .
op _;_ : KindSet KindSet -> KindSet [ctor ditto] .

sorts Equation EquationSet .
subsort Equation < EquationSet .
op eq_=_[_]. : Term Term AttrSet -> Equation [ctor] .
op ceq_=_if_[_]. : Term Term EqCondition AttrSet -> Equation
     [ctor] .
op none : -> EquationSet [ctor] .
op __ : EquationSet EquationSet -> EquationSet
     [ctor assoc comm id: none] .

sorts Rule RuleSet .
subsort Rule < RuleSet .
op rl_=>_[_]. : Term Term AttrSet -> Rule [ctor] .
op crl_=>_if_[_]. : Term Term Condition AttrSet -> Rule [ctor] .
op none : -> RuleSet [ctor] .
op __ : RuleSet RuleSet -> RuleSet [ctor assoc comm id: none] .

sorts StratDecl StratDeclSet .
subsort StratDecl < StratDeclSet .
op strat_:_@_[_]. : Qid TypeList Type AttrSet -> StratDecl [ctor] .
op none : -> StratDeclSet [ctor] .
op __ : StratDeclSet StratDeclSet -> StratDeclSet [ctor assoc comm id: none] .

sorts StratDefinition StratDefSet .
subsort StratDefinition < StratDefSet .
op sd_:=_[_]. : CallStrategy Strategy AttrSet -> StratDefinition [ctor] .
op csd_:=_if_[_]. : CallStrategy Strategy EqCondition AttrSet
      -> StratDefinition [ctor] .
op none : -> StratDefSet [ctor] .
op __ : StratDefSet StratDefSet -> StratDefSet [ctor assoc comm id: none] .
```

For example, we show here the metarepresentations of the modules introduced in Section 5.1 VENDING-MACHINE-SIGNATURE and VENDING-MACHINE.

```
fmod 'VENDING-MACHINE-SIGNATURE is
  nil
  sorts 'Coin ; 'Item ; 'Marking .
  subsort 'Coin < 'Marking .
  subsort 'Item < 'Marking .
  op '__ : 'Marking 'Marking -> 'Marking
       [assoc comm id('null.Marking)] .
  op 'a : nil -> 'Item [format('b! 'o)] .
  op 'null : nil -> 'Marking [none] .
  op '$ : nil -> 'Coin [format('r! 'o)] .
  op 'q : nil -> 'Coin [format('r! 'o)] .
  op 'c : nil -> 'Item [format('b! 'o)] .
  none
  none
endfm

mod 'VENDING-MACHINE is
```

```
  including 'VENDING-MACHINE-SIGNATURE .
  sorts none .
  none
  none
  none
  none
  rl 'M:Marking => '__['M:Marking, 'q.Coin] [label('add-q)] .
  rl 'M:Marking => '__['M:Marking, '$.Coin] [label('add-$)] .
  rl '$.Coin => 'c.Item [label('buy-c)] .
  rl '$.Coin => '__['a.Item, 'q.Coin] [label('buy-a)] .
  rl '__['q.Coin, '__['q.Coin, '__['q.Coin, 'q.Coin]]]
    => '$.Coin [label('change)] .
endm
```

Since `VENDING-MACHINE-SIGNATURE` has no list of imported submodules, no membership axioms, and no equations, those fields are filled, respectively, with the constants `nil` of sort `ImportList`, `none` of sort `MembAxSet`, and `none` of sort `EquationSet`. Similarly, since the module `VENDING-MACHINE` has no subsort declarations and no operator declarations, those fields are filled, respectively, with the constants `none` of sort `SubsortDeclSet` and `none` of sort `OpDeclSet`. Variable declarations are not metarepresented, but rather each variable is metarepresented in its "on the fly"-declaration form, i.e., with its sort or kind.

As mentioned above, parameterized modules are also metarepresented through the notion of a *header*, which is either an identifier (for non-parameterized modules) or an identifier together with a list of parameter declarations (for parameterized modules). Such parameter declarations are metarepresented again with a syntax similar to the user syntax.

```
sorts ParameterDecl NeParameterDeclList ParameterDeclList .
subsorts ParameterDecl < NeParameterDeclList < ParameterDeclList .
op _::_ : Sort ModuleExpression -> ParameterDecl .
op nil : -> ParameterDeclList [ctor] .
op _,_ : ParameterDeclList ParameterDeclList -> ParameterDeclList
     [ctor assoc id: nil prec 121] .
```

Module expressions involving renamings and summations can also be metarepresented with the expected syntax:

```
sort ModuleExpression .
subsort Qid < ModuleExpression .
op _+_ : ModuleExpression ModuleExpression -> ModuleExpression
     [ctor assoc comm] .
op _*(_) : ModuleExpression RenamingSet -> ModuleExpression
     [ctor prec 39 format (d d s n++i n--i d)] .

sorts Renaming RenamingSet .
subsort Renaming < RenamingSet .
op sort_to_ : Qid Qid -> Renaming [ctor] .
op op_to_[_] : Qid Qid AttrSet -> Renaming
     [ctor format (d d d d s d d d)] .
op op_:_->_to_[_] : Qid TypeList Type Qid AttrSet -> Renaming
     [ctor format (d d d d d d d d s d d d)] .
op label_to_ : Qid Qid -> Renaming [ctor] .
op strat_to_ : Qid Qid -> Renaming [ctor] .
op strat_:_@_to_ : Qid TypeList Type Qid -> Renaming [ctor] .
op _,_ : RenamingSet RenamingSet -> RenamingSet
     [ctor assoc comm prec 43 format (d d ni d)] .
```

Finally, the instantiation of a parameterized module is metarepresented as follows:

```
op _{_} : ModuleExpression ParameterList -> ModuleExpression
    [ctor prec 37].

sort EmptyCommaList NeParameterList ParameterList .
subsorts Sort < NeParameterList < ParameterList .
subsort EmptyCommaList < GroundTermList ParameterList .
op empty : -> EmptyCommaList [ctor] .
op _,_ : ParameterList ParameterList -> ParameterList [ctor ditto] .
```

The rules for constructing parameterized metamodules and instantiating parameterized modules existing in the database reflect the object-level rules. In particular, bound parameters are permitted; for example, the following term metarepresents a parameterized module:

```
fmod 'PARMODEX{'X :: 'TRIV} is
  including 'MAP{'String, 'X} .
  sorts 'Foo .
  none
  none
  none
  none
endfm
```

Although, as we will see in the following section, views can be metarepresented as terms of the `View` sort, it is not possible to use the views constructed at the metalevel in module expressions. The views used in the module expressions occurring in metamodules must have been declared at the object level, so that they are present in the database of modules and views declared in the given session. Such views are written in quoted form within metamodule expressions, like `'String` in `'MAP{'String, 'X}` in the example above.

Note that 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.

17.5 The `META-VIEW` module: Metarepresenting views

In the module `META-VIEW`, which imports `META-MODULE`, views are metarepresented in a syntax very similar to their original user syntax.

```
sort View .
op view_from_to_is___endv : Header ModuleExpression ModuleExpression
    SortMappingSet OpMappingSet StratMappingSet -> View [ctor gather (& & & & & &)
    format (d d d d d d d n++i ni ni n--i d)] .
```

The first argument corresponds to the name of the view, while the second and third are module expressions corresponding to the source (usually a theory) and target (usually a module) of the view, respectively. The fourth, fifth and sixth arguments are the sort, operator and strategy mappings defining the view.

The following syntax defines sets of sort mappings in a way completely similar to the user syntax.

```
sorts SortMapping SortMappingSet .
subsort SortMapping < SortMappingSet .
op sort_to_. : Sort Sort -> SortMapping [ctor] .
op none : -> SortMappingSet [ctor] .
```

```
op __ : SortMappingSet SortMappingSet -> SortMappingSet
    [ctor assoc comm id: none format (d ni d)] .
eq S:SortMapping S:SortMapping = S:SortMapping .
```

Analogously, the following syntax is used to define set of operator mappings and strategy mappings.

```
sorts OpMapping OpMappingSet .
subsort OpMapping < OpMappingSet .
op (op_to_.) : Qid Qid -> OpMapping [ctor] .
op (op_:_->_to_.) : Qid TypeList Type Qid -> OpMapping [ctor] .
op (op_to term_.) : Term Term -> OpMapping [ctor] .
op none : -> OpMappingSet [ctor] .
op __ : OpMappingSet OpMappingSet -> OpMappingSet
    [ctor assoc comm id: none format (d ni d)] .
eq O:OpMapping O:OpMapping = O:OpMapping .

sorts StratMapping StratMappingSet .
subsort StratMapping < StratMappingSet .
op (strat_to_.) : Qid Qid -> StratMapping [ctor] .
op (strat_:_@_to_.) : Qid TypeList Type Qid -> StratMapping [ctor] .
op (strat_to-expr_.) : CallStrategy Strategy -> StratMapping [ctor] .
op none : -> StratMappingSet [ctor] .
op __ : StratMappingSet StratMappingSet -> StratMappingSet
    [ctor assoc comm id: none format (d ni d)] .
eq S:StratMapping S:StratMapping = S:StratMapping .
```

Finally, appropriate selectors are used to extract from the metarepresentation of a view the corresponding components, namely, the metarepresentations of its name, of its source, of its target, of its set of sort mappings, and of its set of operator mappings.

```
op getName : View -> Qid .
op getFrom : View -> ModuleExpression .
op getTo : View -> ModuleExpression .
op getSortMappings : View -> SortMappingSet .
op getOpMappings : View -> OpMappingSet .
op getStratMappings : View -> StratMappingSet .
```

For example, the metarepresentation of the view `RingToRat` (see Section 6.3.2) from the theory `RING` to the predefined `RAT` module is as follows:

```
view 'RingToRat from 'RING to 'RAT is
  sort 'Ring to 'Rat .
  op 'z to '0 .
  op 'e.Ring to term 's_['0.Zero] .
  none
endv
```

Then, we can extract some components of this metarepresented view:

```
Maude> reduce in META-VIEW :
         getFrom(view 'RingToRat from 'RING to 'RAT is
                   sort 'Ring to 'Rat .
                   op 'z to '0 .
                   op 'e.Ring to term 's_['0.Zero] .
                   none
                 endv) .
```

```
result Sort: 'RING

Maude> reduce in META-VIEW :
         getOpMappings(view 'RingToRat from 'RING to 'RAT is
                         sort 'Ring to 'Rat .
                         op 'z to '0 .
                         op 'e.Ring to term 's_['0.Zero] .
                         none
                       endv) .
result OpMappingSet:
  op 'z to '0 .
  op 'e.Ring to term 's_['0.Zero] .
```

17.6 The META-LEVEL module: Metalevel operations

The META-LEVEL module, which imports META-VIEW, has several built-in descent functions that provide useful and efficient ways of reducing metalevel computations to object-level ones, as well as several useful operations on sorts and kinds. Since, in general, these operations take among their arguments the metarepresentations of modules, sorts, kinds, terms, and so on, the META-LEVEL modules also provides several built-in functions for moving conveniently between reflection levels. Notice that most of the operations in the module META-LEVEL are partial (as explicitly stated by using the arrow ~> in the corresponding operator declaration). This is due to the fact that they do not make sense on terms that, although may be of the correct sort, for example, Module or Term, either are not correct metarepresentations of modules or are not correct metarepresentations of terms in the module provided as another argument.

Concerning partial operations, the criteria used to choose between using a supersort for the result and having an operator map to a kind is as follows.

If the error return value is built from constructors, say

```
op noParse : Nat -> ResultPair? [ctor] .
op ambiguity : ResultPair ResultPair -> ResultPair? [ctor] .
```

it goes to a supersort. In some sense these are not errors, but merely exceptions or out-of-band results for which there is a carefully defined semantics.

The kind is reserved for nonconstructors which may not be able to reduce at all on illegal arguments, like, for example, in the function (notice the form of the arrow)

```
op metaParse : Module QidList Type? ~> ResultPair? [special (...)] .
```

In this second case, an expression that does not evaluate to the appropriate sort represents a real error.

So, for example, a call to metaParse with an ill-formed module would produce an unreduced term metaParse(...) in the kind, whereas a call to metaParse with valid arguments but a list of tokens that could not be parsed to a term of the desired type in the metamodule would produce a term noParse(...) of sort ResultPair? indicating where the parse first failed.

17.6.1 Moving between reflection levels: upModule, upTerm, downTerm, and others

For a module $\mathcal{R}$ that *has already been loaded* into Maude, the operations upSorts, upSubsortDecl, upOpDecls, upMbs, upEqs, upRls, upStratDecls, upSds, and upModule take as arguments the metarepresentation of the name of $\mathcal{R}$ and a Boolean value b, and return, respectively, the

metarepresentations of the module $\mathcal{R}$, of its sorts, subsort declarations, operator declarations, membership axioms, equations, and rules. If the second argument of these functions is `true`, then the resulting metarepresentations will include the corresponding statements that $\mathcal{R}$ imports from its submodules; but if the second argument is `false`, the resulting metarepresentations will only contain the metarepresentations of the statements explicitly declared in $\mathcal{R}$.

```
op upModule : Qid Bool ~> Module [special (...)] .
op upSorts : Qid Bool ~> SortSet [special (...)] .
op upSubsortDecls : Qid Bool ~> SubsortDeclSet [special (...)] .
op upOpDecls : Qid Bool ~> OpDeclSet [special (...)] .
op upMbs : Qid Bool ~> MembAxSet [special (...)] .
op upEqs : Qid Bool ~> EquationSet [special (...)] .
op upRls : Qid Bool ~> RuleSet [special (...)] .
op upStratDecls : Qid Bool ~> StratDeclSet [special (...)] .
op upSds : Qid Bool ~> StratDefSet [special (...)] .
```

We give below simple examples of using these functions. Note that, since `BOOL` is automatically imported by all modules, its equations are shown when `upEqs` is called with `true` as its second argument. For the same reason, the metarepresentation of the `VENDING-MACHINE-SIGNATURE` module includes an `including` declaration that was not explicit in that module. Here, and in the rest of this section, we assume that the modules `NUMBERS` and `SIEVE` from Chapter 4, as well as the modules `VENDING-MACHINE-SIGNATURE` and `VENDING-MACHINE` from Chapter 5, have already been loaded into Maude.

```
Maude> reduce in META-LEVEL :
        upModule('VENDING-MACHINE-SIGNATURE, false) .
result FModule:
  fmod 'VENDING-MACHINE-SIGNATURE is
  including 'BOOL .
  sorts 'Coin ; 'Item ; 'Marking .
  subsort 'Coin < 'Marking .
  subsort 'Item < 'Marking .
  op '$ : nil -> 'Coin [format('r! 'o)] .
  op '__ : 'Marking 'Marking -> 'Marking
       [assoc comm id('null.Marking)] .
  op 'a : nil -> 'Item [format('b! 'o)] .
  op 'c : nil -> 'Item [format('b! 'o)] .
  op 'null : nil -> 'Marking [none] .
  op 'q : nil -> 'Coin [format('r! 'o)] .
  none
  none
endfm

Maude> reduce in META-LEVEL : upEqs('VENDING-MACHINE, true) .
result EquationSet:
  eq '_and_['true.Bool, 'A:Bool] = 'A:Bool [none] .
  eq '_and_['A:Bool, 'A:Bool] = 'A:Bool [none] .
  eq '_and_['A:Bool, '_xor_['B:Bool, 'C:Bool]]
    = '_xor_['_and_['A:Bool, 'B:Bool], '_and_['A:Bool, 'C:Bool]]
    [none] .
  eq '_and_['false.Bool, 'A:Bool] = 'false.Bool [none] .
  eq '_or_['A:Bool,'B:Bool]
    = '_xor_['_and_['A:Bool, 'B:Bool],'_xor_['A:Bool, 'B:Bool]]
    [none] .
  eq '_xor_['A:Bool, 'A:Bool] = 'false.Bool [none] .
```

```
  eq '_xor_['false.Bool, 'A:Bool] = 'A:Bool [none] .
  eq 'not_['A:Bool] = '_xor_['true.Bool, 'A:Bool] [none] .
  eq '_implies_['A:Bool, 'B:Bool]
    = 'not_['_xor_['A:Bool, '_and_['A:Bool, 'B:Bool]]] [none] .

Maude> reduce in META-LEVEL : upEqs('VENDING-MACHINE, false) .
result EquationSet: (none).EquationSet

Maude> reduce in META-LEVEL : upRls('VENDING-MACHINE, true) .
result RuleSet:
  rl '$.Coin => 'c.Item [label('buy-c)] .
  rl '$.Coin => '__['q.Coin,'a.Item] [label('buy-a)] .
  rl 'M:Marking => '__['$.Coin,'M:Marking] [label('add-$)] .
  rl 'M:Marking => '__['q.Coin,'M:Marking] [label('add-q)] .
  rl '__['q.Coin,'q.Coin,'q.Coin,'q.Coin] => '$.Coin
    [label('change)] .
```

In addition to the `upModule` operator, there is another operator allowing the use of an already loaded module at the metalevel. This operator is defined in the module `META-MODULE` as follows:

```
op [_] : Qid -> Module .
eq [Q:Qid] = (sth Q:Qid is including Q:Qid .
              sorts none . none none none none none none none endsth) .
```

This operator is just syntactic sugar for accessing the corresponding module. Notice that the module is not moved up to the metalevel as `upModule` does, it is just a way of referring to it, and therefore more efficient.

The `META-LEVEL` module also provides a function `upImports` that takes as argument the metarepresentation of the name of a module $\mathcal{R}$. When $\mathcal{R}$ is already in the Maude module database, then `upImports` returns the metarepresentation of its list of imported submodules. The function `upImports` does not take a Boolean argument, as the previous up-functions, since it is not useful to ask for the list of imported submodules of a flattened module.

```
op upImports : Qid ~> ImportList [special (...)] .
```

In the same way, the `META-LEVEL` module provides a function `upView` that takes as argument the metarepresentation of the name of a view; when such a view is in the Maude view database, then `upView` returns the corresponding metarepresentation.

```
op upView : Qid ~> View [special (...)] .
```

As a simple example, let us consider the view `String0` from the predefined theory `DEFAULT` to the predefined module `STRING`, all of them provided in `prelude.maude`; then,

```
Maude> reduce in META-LEVEL : upView('String0) .
result View:
  view 'String0 from 'DEFAULT to 'STRING is
    sort 'Elt to 'String .
    op '0.Elt to term '"".String .
  endv
```

Finally, the `META-LEVEL` module introduces two polymorphic functions. The function `upTerm` takes a term t and returns the metarepresentation of its canonical form. The function `downTerm` takes the metarepresentation of a term t as its first argument and a term t' as its second argument, and returns the canonical form of t, if t is a term in the same kind as t'; otherwise, it returns the canonical form of t'.

```
op upTerm : Universal -> Term [poly (1) special (...)] .
op downTerm : Term Universal -> Universal
     [poly (2 0) special (...)] .
```

As simple examples, we can use the function `upTerm` to obtain the metarepresentation of the term `f(a, f(b, c))` in the module `UP-DOWN-TEST` below, and the function `downTerm` to recover the term `f(a, f(b, c))` from its metarepresentation.

```
fmod UP-DOWN-TEST is
  protecting META-LEVEL .
  sort Foo .
  ops a b c d : -> Foo .
  op f : Foo Foo -> Foo .
  op error : -> [Foo] .
  eq c = d .
endfm

Maude> reduce in UP-DOWN-TEST : upTerm(f(a, f(b, c))) .
result GroundTerm: 'f['a.Foo,'f['b.Foo,'d.Foo]]
```

Notice in the previous example that the given argument has been reduced before obtaining its metarepresentation, more specifically, the subterm `c` has become `d`. In the following examples we can observe the same behavior with respect to `downTerm`.

```
Maude> reduce in UP-DOWN-TEST :
        downTerm('f['a.Foo,'f['b.Foo,'c.Foo]], error) .
result Foo: f(a, f(b, d))

Maude> reduce in UP-DOWN-TEST :
        downTerm(upTerm(f(a, f(b, c))), error) .
result Foo: f(a, f(b, d))
```

In our last example, we show the result of `downTerm` when its first argument does not correspond to the metarepresentation of a term in the module `UP-DOWN-TEST`; notice the constant `e` in the metarepresented term that does not correspond to a declared constant in the module.

```
Maude> reduce in UP-DOWN-TEST :
        downTerm('f['a.Foo,'f['b.Foo,'e.Foo]], error) .
Advisory: could not find a constant e of
         sort Foo in meta-module UP-DOWN-TEST.
result [Foo]: error
```

Due to the failure in moving down the metarepresented term given as first argument, the result is the term given as second argument, namely, `error`, which was declared in the module `UP-DOWN-TEST` as a constant of kind `[Foo]`.

17.6.2 Simplifying: `metaReduce` and `metaNormalize`

`metaReduce`

The (partial) operation `metaReduce` takes as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentation of a term t.

```
sort ResultPair .
op {_,_} : Term Type -> ResultPair [ctor] .
op metaReduce : Module Term ~> ResultPair [special (...)] .
```

When t is a term in $\mathcal{R}$, `metaReduce(`$\overline{\mathcal{R}}$`,`$\bar{t}$`)` returns the metarepresentation of the canonical form of t, using the equations in $\mathcal{R}$, together with the metarepresentation of its corresponding sort or kind. The reduction strategy used by `metaReduce` coincides with that of the `reduce` command (see Sections 4.9 and 23.2).

As said above, in general, when either the first argument of `metaReduce` is a term of sort `Module` but not a correct metarepresentation $\overline{\mathcal{R}}$ of an object module $\mathcal{R}$, or the second argument is not the correct metarepresentation $\bar{t}$ of a term t in $\mathcal{R}$, the operation `metaReduce` is undefined, that is, the term `metaReduce(u,v)` does not reduce and it does not get evaluated to a term of sort `ResultPair`, but only to an expression in the kind `[ResultPair]`.

Appropriate selectors extract from the result pairs their two components:

```
op getTerm : ResultPair -> Term .
op getType : ResultPair -> Type .
```

Using `metaReduce` we can simulate at the metalevel the primes computation example at the end of Section 4.4.7.

```
Maude> reduce in META-LEVEL :
        metaReduce(upModule('SIEVE, false),
          'show_upto_['primes.NatList, 's_^10['0.Zero]]) .
result ResultPair:
  {'_._['s_^2['0.Zero], 's_^3['0.Zero], 's_^5['0.Zero],
        's_^7['0.Zero], 's_^11['0.Zero], 's_^13['0.Zero],
        's_^17['0.Zero], 's_^19['0.Zero], 's_^23['0.Zero],
        's_^29['0.Zero]],
   'NatList}
```

We can also insert a new element into an empty map of the type declared in the module `PARMODEX` at the end of Section 17.4 as follows:

```
Maude> red in META-LEVEL :
        metaReduce(
          fmod 'PARMODEX{'X :: 'TRIV} is
            including 'MAP{'String, 'X} .
            sorts 'Foo .
            none
            none
            none
            none
          endfm,
          'insert['"foo".String, 'A:X$Elt,
                  'empty.Map`{String`,X`}]) .
result ResultPair:
  {'_|->_['"foo".String,'A:X$Elt],'Entry`{String`,X`}}
```

Notice that the module expression `'MAP{'String, 'X}` has a bound parameter `X`, which appears also in the sort `X$Elt` in the on-the-fly declaration of the variable `A:X$Elt`.

`metaNormalize`

The (partial) operation `metaNormalize` takes as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentation of a term t.

```
op metaNormalize : Module Term ~> ResultPair [special (...)] .
```

When t is a term in $\mathcal{R}$, `metaNormalize(`$\overline{\mathcal{R}}$`,`$\bar{t}$`)` returns the metarepresentation of the normal form of t with respect to the equational theory consisting of the equational attributes of the

operators in t, without doing any simplification or rewriting with respect to equations or rules in $\mathcal{R}$, together with the metarepresentation of its corresponding sort or kind. For example, from the declarations in the predefined `NAT` module

```
op s_ : Nat -> NzNat [ctor iter special (...)] .
op _+_ : NzNat Nat -> NzNat [assoc comm prec 33 special (...)] .
op _+_ : Nat Nat -> Nat [ditto] .
```

we know that the successor operator satisfies the `iter` theory (see Section 4.4.2) and that the addition operator is associative and commutative (see Section 4.4.1). With this information it is easy to make sense of the following results:

```
Maude> red in META-LEVEL :
        metaNormalize(upModule('NAT, false), 's_['s_['0.Zero]]) .
result ResultPair: {'s_^2['0.Zero],'NzNat}

Maude> red in META-LEVEL :
        metaNormalize(upModule('NAT, false),
          '_+_['s_['s_['0.Zero]],'0.Zero]) .
result ResultPair: {'_+_['0.Zero,'s_^2['0.Zero]],'NzNat}

Maude> red in META-LEVEL :
        metaNormalize(upModule('NAT, false),
          '_+_['0.Zero,'_+_['s_['s_['0.Zero]],'0.Zero]]) .
result ResultPair: {'_+_['0.Zero,'0.Zero,'s_^2['0.Zero]],'NzNat}
```

Notice that associative terms are flattened and, if they are also commutative, the subterms are sorted with respect to an internal order. Notice also that in the last two examples the subterm `'0.Zero` does not disappear. This is because `0` is not declared as an identity element for `_+_`.

17.6.3 Rewriting: `metaRewrite` and `metaFrewrite`

`metaRewrite`

The (partial) operation `metaRewrite` takes as arguments the metarepresentation of a module $\mathcal{R}$, the metarepresentation of a term t, and a value b of the sort `Bound`, i.e., either a natural number or the constant `unbounded`.

```
sort Bound .
subsort Nat < Bound .
op unbounded :-> Bound [ctor] .
op metaRewrite : Module Term Bound ~> ResultPair [special (...)] .
```

The operation `metaRewrite` is entirely analogous to `metaReduce`, but instead of using only the equational part of a module it now uses both the equations and the rules to rewrite the term. The reduction strategy used by `metaRewrite` coincides with that of the `rewrite` command (see Sections 5.4 and 23.2). That is, the result of `metaRewrite`$(\overline{\mathcal{R}},\ \overline{t},\ b)$ is the metarepresentation of the term obtained from t after at most b applications of the rules in $\mathcal{R}$ using the `rewrite` strategy, together with the metarepresentation of its corresponding sort or kind. When the value `unbounded` is given as the third argument, no bound is imposed to the number of rewrites, and rewriting proceeds to the bitter end.

Using `metaRewrite` we can redo at the metalevel the examples in Section 5.4.

```
Maude> reduce in META-LEVEL :
        metaRewrite(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['$.Coin, '__['q.Coin, 'q.Coin]]], 1) .
```

```
result ResultPair:
  {'__['$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin], 'Marking}

Maude> reduce in META-LEVEL :
        metaRewrite(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['$.Coin, '__['q.Coin, 'q.Coin]]], 2) .
result ResultPair:
  {'__['$.Coin, '$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
       'Marking}
```

`metaFrewrite`

Position fair rewriting, which was described in Section 5.4, is metarepresented by the operation `metaFrewrite`. This (partial) operation takes as arguments the metarepresentation of a module, the metarepresentation of a term, a value of sort `Bound`, and a natural number.

```
op metaFrewrite : Module Term Bound Nat ~> ResultPair
    [special (...)] .
```

The reduction strategy used by `metaFrewrite` coincides with that of the `frewrite` command in Maude, except that a final (semantic) sort calculation is performed at the end in order to produce a correct `ResultPair`. That is, `frewrite`$(\overline{\mathcal{R}},\ \bar{t},\ b,\ n)$ results in the metarepresentation of the term obtained from t after at most b applications of the rules in $\mathcal{R}$ using the `frewrite` strategy, with at most n rewrites at each entitled position on each traversal of a subject term, together with the metarepresentation of its corresponding sort or kind. When the value `unbounded` is given as the third argument, no bound is imposed to the number of rewrites.

Using `metaFrewrite` we can redo at the metalevel the examples in Section 5.4.

```
Maude> reduce in META-LEVEL :
        metaFrewrite(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['$.Coin, '__['q.Coin, 'q.Coin]]],
          1, 1) .
result ResultPair:
  {'__['$.Coin, 'q.Coin, 'q.Coin, 'c.Item], 'Marking}

Maude> reduce in META-LEVEL :
        metaFrewrite(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['$.Coin, '__['q.Coin, 'q.Coin]]],
          12, 1) .
result ResultPair:
  {'__['$.Coin, '$.Coin, '$.Coin, '$.Coin, 'q.Coin, 'q.Coin,
       'q.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'q.Coin,
       'q.Coin,'a.Item,'c.Item],
   'Marking}
```

17.6.4 Applying rules: `metaApply` and `metaXapply`

`metaApply`

The (partial) operation `metaApply` takes as arguments the metarepresentation of a module, the metarepresentation of a term, the metarepresentation of a rule label, the metarepresentation of a set of assignments (possibly empty) defining a partial substitution, and a natural number.

```
sorts Assignment Substitution .
subsort Assignment < Substitution .
op _<-_ : Variable Term -> Assignment [ctor prec 63] .
op none : -> Substitution [ctor] .
op _;_ : Substitution Substitution -> Substitution
    [assoc comm id: none prec 65] .

sort ResultTriple ResultTriple? .
subsort ResultTriple < ResultTriple? .
op {_,_,_} : Term Type Substitution -> ResultTriple [ctor] .
op failure : -> ResultTriple? [ctor] .
op metaApply : Module Term Qid Substitution Nat ~> ResultTriple?
    [special (...)] .
```

The operation `metaApply`$(\overline{\mathcal{R}},\ \bar{t},\ \bar{l},\ \sigma,\ n)$ 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 against all rules with label l in $\mathcal{R}$ 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 `failure` is returned;

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 triple formed by the metarepresentation of the resulting fully reduced term, the metarepresentation of its corresponding sort or kind, and the metarepresentation of the substitution used in the reduction is returned.

The `failure` value should not be confused with the "undefined" value for the `metaApply` operation. As already mentioned before for descent functions in general, this operation is partial because it does not make sense on some nonvalid arguments that are terms of the appropriate sort but are not correct metarepresentations. However, even if all arguments are valid in this sense, the intended rule application may fail, either because there is no match or because the match does not satisfy the corresponding rule condition, and then `failure` is used to represent this situation, which is important to distinguish from ill-formed invocations, for example, for error recovery purposes.

Note also that, according to the information in step 3 above, the last argument of `metaApply` is a natural number used to enumerate (starting from `0`) all the possible solutions of the intended rule application. For efficiency, the different solutions should be generated in order, that is, starting with the argument `0` and increasing it until a failure is obtained, indicating that there are no more solutions.

Appropriate selectors extract from the result triples their three components:

```
op getTerm : ResultTriple -> Term .
op getType : ResultTriple -> Type .
op getSubstitution : ResultTriple -> Substitution .
```

As an example, we can force at the metalevel the rewriting of the term `$` in the module `VENDING-MACHINE`, so that only the rule `buy-c` is used, and only once.

```
Maude> reduce in META-LEVEL :
        metaApply(upModule('VENDING-MACHINE, false),
          '$.Coin, 'buy-c, none, 0) .
result ResultTriple: {'c.Item, 'Item, none}
```

Similarly, we can force the rewriting of the same term so that this time only the rule `add-$` is applied.

```
Maude> reduce in META-LEVEL :
        metaApply(upModule('VENDING-MACHINE, false),
          '$.Coin, 'add-$, none, 0) .
result ResultTriple:
  {'__['$.Coin, '$.Coin], 'Marking, 'M:Marking <- '$.Coin}
```

However, using `metaApply`, we cannot force the term `q $` to be rewritten with the rule `buy-c`, since its lefthand side, `$`, does not match (without extension) this term. In this case, we should use instead the `metaXapply` operation described below.

```
Maude> reduce in META-LEVEL :
        metaApply(upModule('VENDING-MACHINE, false),
          '__['q.Coin, '$.Coin], 'buy-c, none, 0) .
result ResultTriple?: (failure).ResultTriple?
```

`metaXapply`

The (partial) operation `metaXapply` takes as arguments the metarepresentation of a module, the metarepresentation of a term, the metarepresentation of a rule label, the metarepresentation of a set of assignments (possibly empty) defining a partial substitution, a natural number, a `Bound` value, and another natural number.

The operation `metaXapply`$(\overline{\mathcal{R}},\ \bar{t},\ \bar{l},\ \sigma,\ n,\ b,\ m)$ is evaluated as the function `metaApply` but using extension (see Section 4.8) and in any possible position, not only at the top. The arguments n and b can be used to localize the part of the term where the rule application can take place:

- n is the lower bound on depth in terms of nested operators, and should be set to `0` to start searching from the top, while
- the `Bound` argument b indicates the upper bound, and should be set to `unbounded` to have no cut off.

Notice that nested occurrences of an operator with the `assoc` attribute are counted as a single operator for depth purposes, that is, matching takes place on the *flattened term* (see Section 4.8). The same idea applies to `iter` operators (see section 4.4.2): a whole stack of an `iter` operator counts as a single operator. Furthermore, because of matching with extension, the solution may have an extra layer, as illustrated in the matching examples at the end of Section 17.6.5.

The last `Nat` argument m in `metaXapply`$(\overline{\mathcal{R}},\ \bar{t},\ \bar{l},\ \sigma,\ n,\ b,\ m)$, as in the case of the operation `metaApply`, is the solution number, used to enumerate multiple solutions. The first solution is `0`, and they should again be generated in order for efficiency.

The result of `metaXapply` has an additional component, giving the context (a term with a single "hole", represented `[]`) inside the given term t, where the rewriting has taken place. The sort `NeCTermList` represents nonempty lists of terms with exactly one "hole," that is, exactly one term of sort `Context`, the rest being of sort `Term`. The sort `GTermList` is the supersort of `NeCTermList` and `TermList` needed for the `assoc` attribute (hidden in the following declarations in the `ditto` attribute) to make sense.

```
sorts Context NeCTermList GTermList .
subsorts Context < NeCTermList < GTermList .
subsort TermList < GTermList .

op [] : -> Context [ctor] .
op _,_ : TermList NeCTermList -> NeCTermList [ctor ditto] .
op _,_ : NeCTermList TermList -> NeCTermList [ctor ditto] .
op _,_ : GTermList GTermList -> GTermList [ctor ditto] .
op _[_] : Qid NeCTermList -> Context [ctor] .

sorts Result4Tuple Result4Tuple? .
subsort Result4Tuple < Result4Tuple? .
op {_,_,_,_} : Term Type Substitution Context -> Result4Tuple
    [ctor] .
op failure : -> Result4Tuple? [ctor] .

op metaXapply :
    Module Term Qid Substitution Nat Bound Nat ~> Result4Tuple?
    [special (...)] .
```

Appropriate selectors extract from the result 4-tuples their four components:

```
op getTerm : Result4Tuple -> Term .
op getType : Result4Tuple -> Type .
op getSubstitution : Result4Tuple -> Substitution .
op getContext : Result4Tuple -> Context .
```

As an example, we can force at the metalevel the rewriting of the term `$ q` in the module `VENDING-MACHINE` so that only the rule `buy-c` is used (compare with the last `metaApply` example).

```
Maude> reduce in META-LEVEL :
        metaXapply(upModule('VENDING-MACHINE, false),
          '__['q.Coin, '$.Coin], 'buy-c, none, 0, unbounded, 0) .
result Result4Tuple:
  {'__['q.Coin, 'c.Item], 'Marking, none, '__['q.Coin, []]}
```

Notice the fragment `'__['q.Coin, []]` of the result, providing the context where the rule has been applied. Since this is the only possible solution, if we request the "next" solution (by increasing to `1` the last argument), the result will be a failure.

```
Maude> reduce in META-LEVEL :
        metaXapply(upModule('VENDING-MACHINE, false),
          '__['q.Coin, '$.Coin], 'buy-c, none, 0, unbounded, 1) .
result Result4Tuple?: (failure).Result4Tuple?
```

17.6.5 Matching: `metaMatch` and `metaXmatch`

The (partial) operation `metaMatch` takes as arguments the metarepresentation of a module, the metarepresentations of two terms, the metarepresentation of a condition, and a natural number.

```
sort Substitution? .
subsort Substitution < Substitution? .
op noMatch : -> Substitution? [ctor] .
op metaMatch : Module Term Term Condition Nat ~> Substitution?
    [special (...)] .
```

The operation `metaMatch(`$\overline{\mathcal{R}}$`,` $\overline{t}$`,` $\overline{t'}$`,` *Cond*`,` n`)` tries to match at the top the terms t and t' in the module $\mathcal{R}$ in such a way that the resulting substitution satisfies the condition *Cond*. The last argument is used to enumerate possible matches. If the matching attempt is successful, the result is the corresponding substitution; otherwise, `noMatch` is returned. The generalization to `metaXmatch` follows exactly the same ideas as for `metaXapply`. Notice that the operation `metaMatch` provides the metalevel counterpart of the object-level command `match`, while the operation `metaXmatch` provides a generalization of the object-level command `xmatch` (see Sections 4.7, 4.8, and 23.3) in that it is possible to specify min and max depths (in terms of theory layers) and search for proper subterms that do not belong to the top theory layer. The object-level behavior of the `xmatch` command is obtained by setting both min and max depth to 0.

```
sorts MatchPair MatchPair? .
subsort MatchPair < MatchPair? .
op {_,_} : Substitution Context -> MatchPair [ctor] .
op noMatch : -> MatchPair? [ctor] .
op metaXmatch :
    Module Term Term Condition Nat Bound Nat ~> MatchPair?
    [special (...)] .
```

Appropriate selectors extract from the result pairs their two components:

```
op getSubstitution : MatchPair -> Substitution .
op getContext : MatchPair -> Context .
```

In the following examples, we try to match the pattern `M:Marking $` with the term `$ q c a` in several different ways:

- at the top, asking for the first solution,

```
Maude> reduce in META-LEVEL :
        metaMatch(upModule('VENDING-MACHINE, false),
          '__['M:Marking, '$.Coin],
          '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
          nil, 0) .
result Assignment:
  'M:Marking <- '__['q.Coin, 'a.Item, 'c.Item]
```

- at the top, asking for the second solution (that does not exist in this example)

```
Maude> reduce metaMatch(upModule('VENDING-MACHINE, false),
               '__['M:Marking, '$.Coin],
               '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
               nil, 1) .
result Substitution?: (noMatch).Substitution?
```

- anywhere, asking for the first solution,

```
Maude> reduce metaXmatch(upModule('VENDING-MACHINE, false),
               '__['M:Marking, '$.Coin],
               '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
               nil, 0, unbounded, 0) .
result MatchPair:
  {'M:Marking <- '__['q.Coin, 'a.Item, 'c.Item], []}
```

- anywhere, asking for the second solution,

```
Maude> reduce metaXmatch(upModule('VENDING-MACHINE, false),
               '__['M:Marking, '$.Coin],
               '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
               nil, 0, unbounded, 1) .
result MatchPair:
  {'M:Marking <- '__['a.Item, 'c.Item], '__['q.Coin, []]}
```

- at the top, asking for the first solution satisfying a given condition (that again does not exist),

```
Maude> reduce metaMatch(upModule('VENDING-MACHINE, false),
               '__['M:Marking, '$.Coin],
               '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
               'M:Marking = 'a.Item, 0) .
result Substitution?: (noMatch).Substitution?
```

- anywhere, asking for the first solution satisfying a given condition,

```
Maude> reduce metaXmatch(upModule('VENDING-MACHINE, false),
               '__['M:Marking, '$.Coin],
               '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
               'M:Marking = 'a.Item, 0, unbounded, 0) .
result MatchPair:
  {'M:Marking <- 'a.Item, '__['__['q.Coin, 'c.Item], []]}
```

As mentioned in the previous section, when matching with extension, the solution may have an extra layer. Let us consider, for example, the following module:

```
fmod METAXMATCH-EX is
  pr META-LEVEL .
  op foo : QidSet ~> QidSet .
endfm
```

Then we take at the metalevel the pattern `_;_('A, QS:QidSet)` and the (flattened) subject term `foo(_;_('A, 'B, 'C))`, and ask for matches with extension under at most 1 theory layer, as shown in the following reductions:

```
Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
           upTerm(('A ; QS:QidSet)),
           upTerm(foo('A ; 'B ; 'C)), nil, 0, 1, 0) .
result MatchPair: {'QS:QidSet <- '_;_[''B.Sort, ''C.Sort], 'foo[[]]}

Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
           upTerm(('A ; QS:QidSet)),
           upTerm(foo('A ; 'B ; 'C)), nil, 0, 1, 1) .
result MatchPair: {'QS:QidSet <- ''C.Sort, 'foo['_;_[''B.Sort, []]]}

Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
           upTerm(('A ; QS:QidSet)),
           upTerm(foo('A ; 'B ; 'C)), nil, 0, 1, 2) .
result MatchPair: {'QS:QidSet <- ''B.Sort, 'foo['_;_[''C.Sort, []]]}

Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
           upTerm(('A ; QS:QidSet)),
           upTerm(foo('A ; 'B ; 'C)), nil, 0, 1, 3) .
result MatchPair?: (noMatch).MatchPair?
```

Obviously, there is no match at the top, but under one theory layer (the `foo` operator) we have `_;_('A, 'B, 'C)`. The first solution is the expected one, with the variable `QS:QidSet` matching the subterm `_;_('B, 'C)`. However, in the next two solutions we see that we also have the variable `QS:QidSet` matching either the fragment `'C` or `'B` while the other fragment goes into the extension. Then the context in the solution has 2 theory layers but this is just a feature of matching with extension: some solutions will have an extra layer.

As another example of this situation, let us consider the following reductions:

```
Maude> reduce in META-LEVEL :
        metaXmatch(upModule('METAXMATCH-EX, false),
          upTerm(s N:Nat), upTerm(prec(s_^2(0))), nil, 0, 1, 0) .
result MatchPair: {'N:Nat <- 's_['0.Zero], 'prec[[]]}

Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
            upTerm(s N:Nat), upTerm(prec(s_^2(0))), nil, 0, 1, 1) .
result MatchPair: {'N:Nat <- '0.Zero, 'prec['s_[[]]]}
```

Here the context in the first solution has one theory layer while the context in the second has two, but the actual matching problem solved (with extension), namely, `s N <=? s_^2(0)` under the single theory layer provided by the operator `prec` is the same in both reductions.

17.6.6 Searching: `metaSearch` and `metaSearchPath`

`metaSearch`

The operation `metaSearch` takes as arguments the metarepresentation of a module, the metarepresentation of the starting term for search, the metarepresentation of the pattern to search for, the metarepresentation of a condition to be satisfied, the metarepresentation of the kind of search to carry on, a `Bound` value, and a natural number.

```
op metaSearch :
    Module Term Term Condition Qid Bound Nat ~> ResultTriple?
    [special (...)] .
```

The searching strategy used by `metaSearch` coincides with that of the object-level `search` command in Maude (see Sections 5.4 and 23.4). The `Qid` values that are allowed as arguments are: `'*` for a search involving zero or more rewrites (corresponding to `=>*` in the `search` command), `'+` for a search consisting in one or more rewrites (`=>+`), and `'!` for a search that only matches canonical forms (`=>!`). The `Bound` argument indicates the maximum depth of the search, and the `Nat` argument is the solution number. To indicate a search consisting in exactly one rewrite, we set the maximum depth of the search to the number `1`.

Using `metaSearch` we can redo at the metalevel the last example in Section 5.4. The results give us the answer to the question: if I have already inserted one dollar and three quarters in the vending machine, can I get two cakes and an apple? The answer is yes; in fact, there are several ways.

```
Maude> reduce in META-LEVEL :
        metaSearch(upModule('VENDING-MACHINE, false),
          '__['$.Coin, 'q.Coin, 'q.Coin,'q.Coin],
          '__['c.Item, 'a.Item, 'c.Item, 'M:Marking],
          nil, '+, unbounded, 0) .
result ResultTriple:
  {'__['q.Coin,'q.Coin,'q.Coin,'q.Coin,'a.Item,'c.Item,'c.Item],
   'Marking,
   'M:Marking <- '__['q.Coin, 'q.Coin, 'q.Coin, 'q.Coin]}
```

```
Maude> reduce in META-LEVEL :
        metaSearch(upModule('VENDING-MACHINE, false),
          '__['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
          '__['c.Item, 'a.Item, 'c.Item, 'M:Marking],
          nil, '+, unbounded, 1) .
result ResultTriple:
  {'__['a.Item, 'c.Item, 'c.Item],
   'Marking,
   'M:Marking <- 'null.Marking}
```

metaSearchPath

The operation metaSearchPath is complementary to metaSearch and carries out the same search, but instead of returning the final state and matching substitution it returns the sequence of states and rules on a path starting with the reduced initial state and leading to (but not including) the final state.

```
op metaSearchPath :
     Module Term Term Condition Qid Bound Nat ~> Trace?
     [special (...)] .
```

The sort Trace is used to represent the path as a list of triples by means of the following syntax:

```
sorts TraceStep Trace Trace? .
subsorts TraceStep < Trace < Trace? .
op {_,_,_} : Term Type Rule -> TraceStep [ctor] .
op nil : -> Trace [ctor] .
op __ : Trace Trace -> Trace [ctor assoc id: nil format (d n d)] .
op failure : -> Trace? [ctor] .
```

We run again the same two examples as above, with the following results.

```
Maude> reduce in META-LEVEL :
        metaSearchPath(upModule('VENDING-MACHINE, false),
          '__['$.Coin, 'q.Coin, 'q.Coin,'q.Coin],
          '__['c.Item, 'a.Item, 'c.Item, 'M:Marking],
          nil, '+, unbounded, 0) .
result Trace:
  {'__['$.Coin,'q.Coin,'q.Coin,'q.Coin],
   'Marking,
   rl 'M:Marking => '__['$.Coin,'M:Marking] [label('add-$)] .}
  {'__['$.Coin,'$.Coin,'q.Coin,'q.Coin,'q.Coin],
   'Marking,
   rl 'M:Marking => '__['$.Coin,'M:Marking] [label('add-$)] .}
  {'__['$.Coin,'$.Coin,'$.Coin,'q.Coin,'q.Coin,'q.Coin],
   'Marking,
   rl '$.Coin => 'c.Item [label('buy-c)] .}
  {'__['$.Coin,'$.Coin,'q.Coin,'q.Coin,'q.Coin,'c.Item],
   'Marking,
   rl '$.Coin => 'c.Item [label('buy-c)] .}
  {'__['$.Coin,'q.Coin,'q.Coin,'q.Coin,'c.Item,'c.Item],
   'Marking,
   rl '$.Coin => '__['q.Coin,'a.Item] [label('buy-a)] .}
```

```
Maude> reduce in META-LEVEL :
        metaSearchPath(upModule('VENDING-MACHINE, false),
          '__['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
          '__['c.Item, 'a.Item, 'c.Item, 'M:Marking],
          nil, '+, unbounded, 1) .
result Trace:
  {'__['$.Coin,'q.Coin,'q.Coin,'q.Coin],
   'Marking,
   rl 'M:Marking => '__['$.Coin,'M:Marking] [label('add-$)] .}
  {'__['$.Coin,'$.Coin,'q.Coin,'q.Coin,'q.Coin],
   'Marking,
   rl '$.Coin => 'c.Item [label('buy-c)] .}
  {'__['$.Coin,'q.Coin,'q.Coin,'q.Coin,'c.Item],
   'Marking,
   rl '$.Coin => '__['q.Coin,'a.Item] [label('buy-a)] .}
  {'__['q.Coin,'q.Coin,'q.Coin,'q.Coin,'a.Item,'c.Item],
   'Marking,
   rl '__['q.Coin,'q.Coin,'q.Coin,'q.Coin] => '$.Coin
     [label('change)] .}
  {'__['$.Coin,'a.Item,'c.Item],
   'Marking,
   rl '$.Coin => 'c.Item [label('buy-c)] .}
```

The operations `metaSearchPath` and `metaSearch` share caching, so calling one after the other on the same arguments only performs a single search.

17.6.7 Rewriting using strategies: `metaSrewrite`

The operation `metaSrewrite` rewrites a metaterm according to a metastrategy in a given metamodule.

```
op metaSrewrite :
    Module Term Strategy SrewriteOption Nat ~> ResultPair? [special (...)] .
```

It is the metarepresentation of the `srewrite` and `dsrewrite` commands, depending on whether the `SrewriteOption` parameter is `breadthFirst` or `depthFirst` respectively. Like similar descent functions, the last parameter allows enumerating the strategy solutions, and evaluates to the constant `failure` when the given index is higher than the number of solutions.

For example, in the `QUEENS-STRAT` module of Section 10.3, we can obtain all possible ways of extending the partial solution `1 3 5`, by enumerating the solutions of the `expand` strategy defined in `QUEENS-STRAT`:

```
Maude> red in META-LEVEL : metaSrewrite(['QUEENS-STRAT], upTerm(1 3 5),
                                        'expand[[empty]], breadthFirst, 0) .
result ResultPair:
{'__['s_['0.Zero],'s_^3['0.Zero],'s_^5['0.Zero],'s_^2['0.Zero]],'NeList`{Nat`}}

Maude> red metaSrewrite(['QUEENS-STRAT], upTerm(1 3 5),
                        'expand[[empty]], breadthFirst, 1) .
result ResultPair:
{'__['s_['0.Zero],'s_^3['0.Zero],'s_^5['0.Zero],'s_^7['0.Zero]],'NeList`{Nat`}}

Maude> red metaSrewrite(['QUEENS-STRAT], upTerm(1 3 5),
                        'expand[[empty]], breadthFirst, 2) .
result ResultPair:
```

```
{'__['s_['0.Zero],'s_^3['0.Zero],'s_^5['0.Zero],'s_^8['0.Zero]],'NeList`{Nat`}}

Maude> red metaSrewrite(['QUEENS-STRAT], upTerm(1 3 5),
                        'expand[[empty]], breadthFirst, 3) .
result ResultPair?:
failure
```

17.6.8 Unification: `metaUnify`, `metaIrredundantUnify`, `metaDisjointUnify` and `metaIrredundantDisjointUnify`

The unification command of Section 13.4 is reflected in the `META-LEVEL` module by two descent functions:

```
op metaUnify :
     Module UnificationProblem Nat Nat ~> UnificationPair? [special (...)] .

op metaDisjointUnify :
     Module UnificationProblem Nat Nat ~> UnificationTriple? [special (...)] .
```

These two metalevel functions work on unification problems constructed by means of the following signature:

```
sorts UnificandPair UnificationProblem .
subsort UnificandPair < UnificationProblem .
op _=?_ : Term Term -> UnificandPair [ctor prec 71] .
op _/\_ : UnificationProblem UnificationProblem -> UnificationProblem
     [ctor assoc comm prec 73] .
```

The key difference between `metaUnify` and `metaDisjointUnify` is that the latter assumes that the variables in the left and righthand unificands are to be considered *disjoint* even when they are not so, and it generates each solution to the given unification problem not as a single substitution, but as a *pair* of substitutions, one for left unificands and the other for right unificands. This functionality is very useful for applications, such as critical-pair checking or narrowing, where a disjoint copy of the terms or rules involved must always be computed before unification is performed. Indeed, what the `metaDisjointUnify` operation avoids is precisely the need for explicitly computing such disjoint copies. The need for two substitutions in each solution is then obvious, since the terms in the given unification problem need not be made explicitly disjoint, but their (accidentally) common variables must be treated differently, as if they were disjoint.

Since it is convenient to *reuse* variable names from unifiers in new problems, for example in narrowing, this is allowed via the third argument, which is the largest number n appearing in a unificand variable of the form `#n:Sort` (see Section 13.4). The latest version of Maude includes an alternative interface to variable reuse by using a `Qid` instead of a natural number in the third argument, which is the identifier ξ used in unificand variables of the form ξ`n:Sort` .

```
op metaUnify :
     Module UnificationProblem Qid Nat ~> UnificationPair? [special (...)] .

op metaDisjointUnify :
     Module UnificationProblem Qid Nat ~> UnificationTriple? [special (...)] .
```

This avoids variable name clashes with the previous variable families `'#`, `'%` and `'@` provided for, respectively, unification modulo axioms in Chapter 13, variant unification in Chapter 14, and narrowing in Chapter 15.

When we are interested in the minimal set of most general unifiers modulo axioms, we should use the following two descent functions, which are defined only for the alternative of using a `Qid` instead of a natural number in the third argument:

```
op metaIrredundantUnify :
    Module UnificationProblem Qid Nat ~> UnificationPair? [special (...)] .

op metaIrredundantDisjointUnify :
    Module UnificationProblem Qid Nat ~> UnificationTriple? [special (...)] .
```

As is usual for descent functions, the last argument in the function is used to select which result is wanted, starting from 0. Caching is used so that if unifier i has just been returned, requesting unifier $i + 1$ gives rise to an incremental computation.

Results are returned using the following constructors when a natural number is used for the third argument:

```
subsort UnificationPair < UnificationPair? .
subsort UnificationTriple < UnificationTriple? .
op {_,_} : Substitution Nat -> UnificationPair [ctor] .
op {_,_,_} : Substitution Substitution Nat -> UnificationTriple [ctor] .
```

as appropriate for the descent function. The final `Nat` component is the largest n occurring in a fresh metavariable of the form *`#n:Sort`*. In this way, when we want to reuse variable names from unifiers, the next invocation of the function can use this parameter to make sure that the new variables generated are always fresh.

In the case where a call to `metaUnify` or `metaDisjointUnify` is invoked with a `Qid` in the third argument, the following different constructors are used:

```
op {_,_} : Substitution Qid -> UnificationPair [ctor] .
op {_,_,_} : Substitution Substitution Qid -> UnificationTriple [ctor] .
```

and now they return the `Qid` specifying a variable family instead of the largest natural number used. Note that such `Qid` will indeed be different to the `Qid` given in the third argument.

When no unifier with a given index exists, the constant

```
op noUnifier : -> UnificationPair? [ctor] .
```

or, respectively, the constant

```
op noUnifier : -> UnificationTriple? [ctor] .
```

is returned as appropriate for the corresponding descent function.

Recall that for unification modulo associative symbols no finite set of unifiers may exist, yet a finite set is returned with a warning if the set may be incomplete (see Section 13.4.6). At the metalevel, the role of this warning is played by the constant:

```
op noUnifierIncomplete : -> UnificationPair? [ctor] .
```

or, respectively, the constant

```
op noUnifierIncomplete : -> UnificationTriple? [ctor] .
```

which is returned when a finite set of most general unifiers cannot be ensured.

We can illustrate the use of these metalevel functions with a few examples. The first one comes from the previous section, but moved up at the metalevel:

```
Maude> reduce in META-LEVEL :
        metaUnify(upModule('UNIFICATION-EX1, false),
          'f['X:Nat, 'Y:NzNat] =? 'f['Z:NzNat, 'U:Nat] /\
```

```
            'V:NzNat =? 'f['X:Nat, 'U:Nat], 0, 0) .

result UnificationPair:
      {'U:Nat <- '#1:NzNat ;
       'V:NzNat <- 'f['#2:NzNat, '#1:NzNat] ;
       'X:Nat <- '#2:NzNat ;
       'Y:NzNat <- '#1:NzNat ;
       'Z:NzNat <- '#2:NzNat, 2}
```

The second example shows that we can request fresh variables with arbitrarily large numbering:

```
Maude> reduce in META-LEVEL :
        metaUnify(upModule('NAT, false),
          '_+_['X:Nat,'Y:Nat] =? '_+_['A:Nat,'B:Nat],
           10000000000000000000, 0) .

result UnificationPair:
      {'A:Nat <- '_+_['#10000000000000000001:Nat,
                     '#10000000000000000002:Nat] ;
       'B:Nat <- '_+_['#10000000000000000003:Nat,
                     '#10000000000000000004:Nat] ;
       'X:Nat <- '_+_['#10000000000000000001:Nat,
                     '#10000000000000000003:Nat] ;
       'Y:Nat <- '_+_['#10000000000000000002:Nat,
                     '#10000000000000000004:Nat],
       10000000000000000004}
```

The following example shows a similar unification problem but with much smaller numberings in fresh variables, and now involving an invocation of `metaDisjointUnify`.

```
Maude> reduce in META-LEVEL :
        metaDisjointUnify(upModule('NAT, false),
          '_+_['X:Nat, 'Y:Nat] =? '_+_['X:Nat, 'B:Nat], 0, 0) .

result UnificationTriple: {
       'X:Nat <- '_+_['#1:Nat, '#2:Nat] ;
       'Y:Nat <- '_+_['#3:Nat, '#4:Nat],
       'B:Nat <- '_+_['#1:Nat, '#3:Nat] ;
       'X:Nat <- '_+_['#2:Nat, '#4:Nat], 4}
```

Yet another example shows how using variable names in unification problems with larger numbers than declared by the third argument generates a warning and no reduction.

```
Maude> reduce in META-LEVEL :
        metaUnify(upModule('NAT, false),
          '_+_['X:Nat,'Y:Nat] =? '_+_['#1:Nat,'Y:Nat], 0, 0) .

Warning: unsafe variable name #1:Nat in unification problem.

result [UnificationPair?]:
  metaUnify(th 'NAT is
             including 'NAT .
             sorts none .
             none
             none
```

```
            none
            none
            none
          endth,
          '_+_['X:Nat, 'Y:Nat] =? '_+_['#1:Nat, 'Y:Nat], 0, 0)
```

Similarly, the following example shows how using a variable of the form ξ`n:Sort` for ξ being a different `Qid` to that given in the third argument generates a warning and no reduction.

```
Maude> reduce in META-LEVEL :
        metaUnify(upModule('NAT, false),
          '_+_['X:Nat,'Y:Nat] =? '_+_['#1:Nat,'Y:Nat], '%, 0) .

Warning: unsafe variable name #1:Nat in unification problem.

result [UnificationPair?]:
  metaUnify(th 'NAT is
            including 'NAT .
            sorts none .
            none
            none
            none
            none
            none
          endth,
          '_+_['X:Nat, 'Y:Nat] =? '_+_['#1:Nat, 'Y:Nat], '%, 0)
```

And finally an example of incomplete unification for the associative case. If we move to the metalevel the unification problem with an infinite set of most general unifiers, `0 : X` $=^?$ `X : 0`, we get the first unifier of the family:

```
Maude> reduce in META-LEVEL :
    metaUnify(upModule('UNIFICATION-EX4, true),
    '_:_['0.Nat,'X:NList] =? '_:_['X:NList,'0.Nat], 0, 0) .

Warning: Unification modulo the theory of operator _:_ has encountered
an instance for which it may not be complete.

result UnificationPair: {
       'X:NList <- '0.Zero,0}
```

but successive calls for unifiers get the constant `noUnifierIncomplete`:

```
Maude> reduce in META-LEVEL :
     metaUnify(upModule('UNIFICATION-EX4, true),
     '_:_['0.Nat,'X:NList] =? '_:_['X:NList,'0.Nat], 0, 1) .

result UnificationPair?: (noUnifierIncomplete).UnificationPair?
```

Note that we got the constant `noUnifierIncomplete` instead of the `noUnifier` constant, which is the output for the case of a finitary set of most general unifiers.

Several auxiliary functions have been defined by equations, allowing easy extraction of information.

```
op getSubstitution : UnificationPair -> Substitution .
op getVariableFamily : UnificationPair -> Qid .
op getLhsSubstitution : UnificationTriple -> Substitution .
```

```
op getRhsSubstitution : UnificationTriple -> Substitution .
op getVariableFamily : UnificationTriple -> Qid .
```

Since it is quite common to apply a substitution to a term, we have included such feature as a function defined by equations

```
op applySubstitution : Module Term Substitution -> Term .
```

17.6.9 Variants: metaGetVariant

The procedure for variant generation of Section 14.4 is also available at the metalevel of Maude thanks to the metaGetVariant and metaGetIrredundantVariant functions provided in the META-LEVEL module.

```
op metaGetVariant : Module Term TermList Nat Nat ~> Variant?
     [special (...)] .
op metaGetIrredundantVariant : Module Term TermList Nat Nat ~> Variant?
     [special (...)] .
```

The third argument allows a list of irreducible terms, see Section 14.5 for details. As in Section 17.6.8, it is convenient to *reuse* variable names from terms; this is allowed via the fourth argument, which is the largest number n appearing in fresh variables of the form *#n:Sort* or *%n:Sort*. The latest version of Maude includes an alternative interface to these two functions using a Qid instead of a natural number in the fourth argument, which is the identifier ξ used in unificand variables of the form ξ*n:Sort*.

```
op metaGetVariant : Module Term TermList Qid Nat ~> Variant?
     [special (...)] .
op metaGetIrredundantVariant : Module Term TermList Qid Nat ~> Variant?
     [special (...)] .
```

This avoids variable name clashes with the previous variable families '#, '% and '@ provided for, respectively, unification modulo axioms in Chapter 13, variant unification in Chapter 14, and narrowing in Chapter 15.

As usual for descent functions, the last argument in the function is used to select which result is wanted, starting from 0. Caching is used so that if variant i has just been returned, requesting unifier $i + 1$ gives rise to an incremental computation.

The result sort is defined by means of the following data:

```
sorts Variant Variant? .
subsort Variant < Variant? .
op {_,_,_,_,_} : Term Substitution Nat Parent Bool -> Variant [ctor] .
op noVariant : -> Variant? [ctor] .
op noVariantIncomplete : -> Variant? [ctor] .
```

Again, the third argument denotes the largest number n used in the fresh variables appearing in the solutions. The fourth and fifth arguments are useful for applications based on the execution narrowing tree rather than the set of variants, see the example below.

In the case where a call to metaGetVariant or metaGetIrredundantVariant is invoked with a Qid in the fourth argument, the following different constructor is used:

```
op {_,_,_,_,_} : Term Substitution Qid Parent Bool -> Variant [ctor] .
```

and now it returns the Qid specifying a variable family instead of the largest natural number used. Note that such Qid will indeed be different to the Qid given in the fourth argument.

We can illustrate the use of this metalevel function with the variant generation of the configuration < $ q q X:Marking Y:Marking> for the first variant.

```
Maude> reduce in META-LEVEL :
   metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
   '<_>['__['$.Coin,'q.Coin,'q.Coin,'X:Marking,'Y:Marking]], empty, 0, 0) .
result Variant: {'<_>['__['$.Coin,'q.Coin,'q.Coin,'#1:Marking,'#2:Marking]],
   'X:Marking <- '#1:Marking ;
   'Y:Marking <- '#2:Marking,2,none,false}
```

Then the second possible variant:

```
Maude> reduce in META-LEVEL :
   metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
   '<_>['__['$.Coin,'q.Coin,'q.Coin,'X:Marking,'Y:Marking]], empty, 0, 1) .
result Variant: {'<_>['__['$.Coin,'$.Coin,'%1:Marking,'%2:Marking]],
   'X:Marking <- '__['q.Coin,'q.Coin,'%1:Marking] ;
   'Y:Marking <- '%2:Marking,2,0,true}
```

Then the third possible variant:

```
Maude> reduce in META-LEVEL :
   metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
   '<_>['__['$.Coin,'q.Coin,'q.Coin,'X:Marking,'Y:Marking]], empty, 0, 2) .
result Variant: {'<_>['__['$.Coin,'$.Coin,'%1:Marking,'%2:Marking]],
   'X:Marking <- '__['q.Coin,'%1:Marking] ;
   'Y:Marking <- '__['q.Coin,'%2:Marking],2,0,true}
```

Then the fourth possible variant:

```
Maude> reduce in META-LEVEL :
   metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
   '<_>['__['$.Coin,'q.Coin,'q.Coin,'X:Marking,'Y:Marking]], empty, 0, 3) .
result Variant: {'<_>['__['$.Coin,'$.Coin,'%1:Marking,'%2:Marking]],
   'X:Marking <- '%1:Marking ;
   'Y:Marking <- '__['q.Coin,'q.Coin,'%2:Marking],2,0,false}
```

Then the fifth possible variant:

```
Maude> reduce in META-LEVEL :
   metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
   '<_>['__['$.Coin,'q.Coin,'q.Coin,'X:Marking,'Y:Marking]], empty, 0, 4) .
result Variant: {'<_>['__['$.Coin,'$.Coin,'$.Coin,'#1:Marking,'#2:Marking]],
   'X:Marking <- '__['q.Coin,'q.Coin,'q.Coin,'#1:Marking] ;
   'Y:Marking <- '__['q.Coin,'q.Coin,'q.Coin,'#2:Marking],2,1,false}
```

And there are no more variants.

```
Maude> reduce in META-LEVEL :
   metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
   '<_>['__['$.Coin,'q.Coin,'q.Coin,'X:Marking,'Y:Marking]], empty, 0, 5) .
result Variant?: noVariant
```

Using the fourth and fifth arguments of each returned variant, we can reconstruct the execution narrowing tree of Figure 17.2. The fourth argument of each variant is the identifier of the parent variant; the identifier of each variant is indeed the last argument of its associated call to `metaGetVariant`. The fifth argument is a Boolean: true meaning that there is at least one other variant in that level of the narrowing tree, and false meaning that this is the last one in that level of the narrowing tree. Note that variants return the whole composed substitution and the intermediate unifier shown in Figure 17.2 between variants 1 and 4 has to be extracted manually.

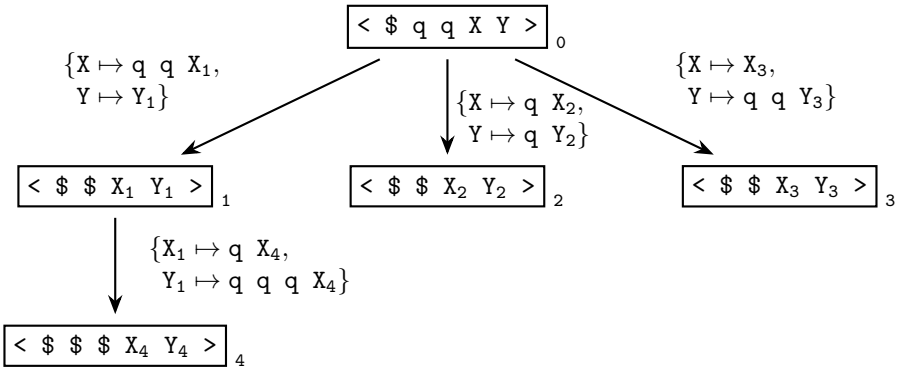


Figure 17.2: Folding variant narrowing tree for the term `< $ q q X Y >`.

We can reproduce the example of the vending machine using irreducible terms showed in Section 14.5 as follows.

```
Maude> reduce in META-LEVEL :
   metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
   '<_>['__['$.Coin,'q.Coin,'q.Coin,'X:Marking,'Y:Marking]],
   ('__['q.Coin,'q.Coin,'X:Marking],'__['q.Coin.'X:Marking], 'X:Marking),
   0, 0) .
result Variant: {'<_>['__['$.Coin,'q.Coin,'q.Coin,'#1:Marking,'#2:Marking]],
   'X:Marking <- '#1:Marking ;
   'Y:Marking <- '#2:Marking,2,none,false}
```

The command prints only three of the five variants returned by the previous call without any irreducible term. The variants where `X:Marking` is mapped to `q q %1:Marking` or `q q q %1:Marking` are discarded, since they violate the condition that `q q X:Marking` must be irreducible under substitution.

```
Maude> reduce in META-LEVEL :
   metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
   '<_>['__['$.Coin,'q.Coin,'q.Coin,'X:Marking,'Y:Marking]],
   ('__['q.Coin,'q.Coin,'X:Marking],'__['q.Coin.'X:Marking], 'X:Marking),
   0, 3) .
result Variant?: noVariant
```

Let us also show an example of an incomplete variant generation at the metalevel. If we move to the metalevel the incomplete variant generation for term `duplicate(prefix(L) : tail(L))` of Section 14.7, we get the first variant:

```
Maude> reduce in META-LEVEL :
    metaGetVariant(upModule('VARIANT-UNIFICATION-ASSOC, true),
       'duplicate['_:_['prefix['L:NList],'tail['L:NList]]], empty, 0, 0) .
result Variant: {'duplicate['_:_['prefix['#1:NList],'tail['#1:NList]]],
       'L:NList <- '#1:NList,1,none,false}
```

And when we ask for the eighth variant, we get the constant `noVariantIncomplete`:

```
Maude> reduce in META-LEVEL :
    metaGetVariant(upModule('VARIANT-UNIFICATION-ASSOC, true),
    'duplicate['_:_['prefix['L:NList],'tail['L:NList]]], empty, 0, 7) .
result Variant?: noVariantIncomplete
```

Note that we got the constant `noVariantIncomplete` instead of the `noVariant` constant, which is the output for the case of a finitary set of variants.

17.6.10 Variant Matching and Unification: `metaVariantUnify`, `metaVariantDisjointUnify`, and `metaVariantMatch`

The procedure for variant-based equational unification of Section 14.9 is also available at the metalevel by means of the following functions provided in the `META-LEVEL` module.

```
op metaVariantUnify :
     Module UnificationProblem TermList Nat Nat ~> UnificationPair?
     [special (...)] .

op metaVariantDisjointUnify :
     Module UnificationProblem TermList Nat Nat ~> UnificationTriple?
     [special (...)] .
```

The unification problems and the result sort are the same as in Section 17.6.8. The third argument allows a list of irreducible terms, see Section 14.10 for details.

```
sorts UnificandPair UnificationProblem .
subsort UnificandPair < UnificationProblem .
op _=?_ : Term Term -> UnificandPair [ctor prec 71] .
op _/\_ : UnificationProblem UnificationProblem -> UnificationProblem
     [ctor assoc comm prec 73] .

subsort UnificationPair < UnificationPair? .
subsort UnificationTriple < UnificationTriple? .
op {_,_} : Substitution Nat -> UnificationPair [ctor] .
op {_,_,_} : Substitution Substitution Nat -> UnificationTriple [ctor] .
```

The latest version of Maude includes an alternative interface to these two functions using a `Qid` instead of a natural number in the fourth argument, which is the identifier ξ used in unificand variables of the form ξ`n:Sort`.

```
op metaVariantUnify :
     Module UnificationProblem TermList Qid VariantOptionSet Nat ~> UnificationPair?
     [special (...)] .

op metaVariantDisjointUnify :
     Module UnificationProblem TermList Qid VariantOptionSet Nat ~> UnificationTriple?
     [special (...)] .
```

In the case where a call to `metaVariantUnify` or `metaVariantDisjointUnify` is invoked with a `Qid` in the fourth argument, the following different constructors are used:

```
op {_,_} : Substitution Qid -> UnificationPair [ctor] .
op {_,_,_} : Substitution Substitution Qid -> UnificationTriple [ctor] .
```

and now they return the `Qid` specifying a variable family instead of the largest natural number used. Note that such `Qid` will indeed be different to the `Qid` given in the third argument.

When we are interested in the minimal set of most general unifiers, there is no alternative command, as in Section 17.6.8, and the option `filter` must be used in the fifth argument of the commands:

```
sorts VariantOption VariantOptionSet .
subsort VariantOption < VariantOptionSet .
```

```
ops delay filter : -> VariantOption [ctor] .
op none : -> VariantOptionSet [ctor] .
op __ : VariantOptionSet VariantOptionSet -> VariantOptionSet
    [ctor assoc comm id: none] .
```

Note that the version of the command with a counter for new variables does not include this fifth argument for options and, thus, cannot return the minimal set of unifiers.

We can illustrate the use of this metalevel function with the variant unification of the two terms of Section 14.9: `< q q X:Marking >` and `< $ Y:Marking >`:

```
Maude> reduce in META-LEVEL :
   metaVariantUnify(upModule('VARIANT-VENDING-MACHINE, true),
      '<_>['__['q.Coin,'q.Coin,'X:Marking]] =?
        '<_>['__['$.Coin,'Y:Marking]],
      empty, '@, 0) .
result UnificationPair: {
  'X:Marking <- '__['$.Coin,'%1:Marking] ;
  'Y:Marking <- '__['q.Coin,'q.Coin,'%1:Marking],'%}
```

Let us also illustrate the use of incomplete variant unification by moving to the metalevel the incomplete unification problem of Section 14.12: `head(L:NList) =? last(L:NList) /\ prefix(L:NList) =? tail(L:NList)`.

```
Maude> reduce in META-LEVEL :
    metaVariantUnify(upModule('VARIANT-UNIFICATION-ASSOC, true),
    'head['L:NList] =? 'last['L:NList] /\
    'prefix['L:NList] =? 'tail['L:NList], empty, 0, 0) .

Warning: Unification modulo the theory of operator _:_ has encountered
an instance for which it may not be complete.

result UnificationPair: {
       'L:NList <- '_:_['%1:Nat,'%1:Nat,'%1:Nat],1}
```

And when we try to obtain the third unifier we get the constant `noUnifierIncomplete`.

```
Maude> reduce in META-LEVEL :
 metaVariantUnify(upModule('VARIANT-UNIFICATION-ASSOC, true),
 'head['L:NList] =? 'last['L:NList] /\
 'prefix['L:NList] =? 'tail['L:NList], empty, 0, 2) .

result UnificationPair?: (noUnifierIncomplete).UnificationPair?
```

Several auxiliary functions have been defined by equations, allowing easy extraction of information.

```
op getTerm : Variant -> Term .
op getSubstitution : Variant -> Substitution .
op getVariableFamily : Variant -> Qid .
op getParent : Variant -> Parent .
op getMoreVariantsInLayerFlag : Variant -> Bool .
```

The procedure for variant-based equational matching of Section 14.13 is also available at the metalevel by means of the following function provided in the `META-LEVEL` module.

```
op metaVariantMatch :
    Module MatchingProblem TermList Qid VariantOptionSet Nat ~> Substitution?
```

The new matching problems are as follows.

```
sorts PatternSubjectPair MatchingProblem .
subsort PatternSubjectPair < MatchingProblem .
op _<=?_ : Term Term -> PatternSubjectPair [ctor prec 71] .
op _/\_ : MatchingProblem MatchingProblem -> MatchingProblem [ctor assoc comm prec 73] .
```

We can illustrate the use of this metalevel function with the variant matching of the two terms of Section 14.13: `< q q X:Marking >` and `< $ Y:Marking >`:

```
Maude> reduce in META-LEVEL :
    metaVariantMatch(upModule('VARIANT-VENDING-MACHINE, true),
    '<_>['__['q.Coin,'q.Coin,'X:Marking]] <=?
    '<_>['__['$.Coin,'Y:Marking]],
    empty, '@, none, 0) .
result Assignment:
  'X:Marking <- '__['q.Coin,'q.Coin,'Y:Marking]
```

Note that the `delay` and `filter` constants of the sort `VariantOptionSet` have no effect for variant matching and we should always use `none`.

17.6.11 Narrowing: `metaNarrowingApply`, `metaNarrowingSearch` and `metaNarrowingSearchPath`

Narrowing is also available at the metalevel by using the functions `metaNarrowingApply`, `metaNarrowingSearch` and `metaNarrowingSearchPath`. Note that there is no user level command associated to the metalevel function `metaNarrowingApply`.

The invocation of just one narrowing step is reproduced by function `metaNarrowingApply`:

```
op metaNarrowingApply :
   Module Term TermList Qid Nat -> NarrowingApplyResult?
   [special ...] .
```

The result sort is defined by means of the following data:

```
sorts NarrowingApplyResult NarrowingApplyResult? .
subsort NarrowingApplyResult < NarrowingApplyResult? .

 op {_,_,_,_,_,_,_} : Term Type Context Qid Substitution Substitution Qid
                   -> NarrowingApplyResult
      [ctor format (d n++i d d d ni d ni d d d d d ni n--i d)] .
op failure : -> NarrowingApplyResult? [ctor] .
op failureIncomplete : -> NarrowingApplyResult? [ctor] .
```

The third argument allows a list of irreducible terms, as in Sections 17.6.9 and 17.6.10. The fourth argument provides the identifier ξ used in variables of the form ξ*n:Sort* appearing in the given term, again as in Sections 17.6.9 and 17.6.10. The last argument is the chosen narrowing step. If there is no solution, the `failure` constant is returned if no incompleteness situation related to associative unification has been found; otherwise, the `failureIncomplete` constant is returned.

For the `NARROWING-VENDING-MACHINE` system module introduced at the beginning of Section 15.6, the following one-step narrowing command can be given

```
Maude> reduce in META-LEVEL :
metaNarrowingApply(upModule('NARROWING-VENDING-MACHINE, false),
'<_>['M:Money], empty, '@, 0) .
```

```
result NarrowingApplyResult: {
  '<_>['__['a.Item,'q.Coin,'%1:Money]],'State,
  [],
  'buy-a,
    'M:Money <- '__['$.Coin,'%1:Money],
    'M:Marking <- '%1:Money,
  '%
}
```

Note that two substitutions are returned, one for the input term and another for the lefthand side of the applied rule.

The narrowing-based reachability analysis of Section 15.6 is available at the metalevel by using the function `metaNarrowingSearch`:

```
op metaNarrowingSearch :
   Module Term Term Qid Bound Qid Nat -> NarrowingSearchResult?
   [special ...] .
```

The result sort is defined by means of the following data:

```
sorts NarrowingSearchResult NarrowingSearchResult? .
subsort NarrowingSearchResult < NarrowingSearchResult? .

op {_,_,_,_,_,_} : Term Type Substitution Qid Substitution Qid
                -> NarrowingSearchResult
     [ctor format (d n++i d d d d d ni d d d ni n--i d)] .
op failure : -> NarrowingSearchResult? [ctor] .
op failureIncomplete : -> NarrowingSearchResult? [ctor] .
```

The first `Qid` argument metarepresents the appropriate search arrow, similar to the `metaSearch` command (see Section 17.6.6). The second `Qid` determines whether folding is applied or not, see Section 15.7. The constant `'none` indicates that standard narrowing without any folding is applied, as the `vu-narrow` command of Section 15.6. The constant `'match` indicates that folding narrowing is applied, as the `fvu-narrow` command of Section 15.7. For the bounds, the `Bound` one is the maximum length of the narrowing sequences, whereas the `Nat` is the chosen solution (in order to provide all solutions in a sequential way, as many metalevel commands in Maude do). If there is no solution, the `failure` constant is returned if no incompleteness situation related to associative unification has been found; otherwise, the `failureIncomplete` constant is returned.

For the `NARROWING-VENDING-MACHINE` system module introduced at the beginning of Section 15.6, the following `search` command considered above

```
Maude> vu-narrow [1] in NARROWING-VENDING-MACHINE : < M:Money > =>* < a c > .
```

can be specified at the metalevel as follows, where `'<_>['M:Money]` is the metarepresentation of the state `< M:Money >`, `'<_>['__['a.Item,'c.Item]]` is the metarepresentation of the state `< a c >`, and we use the coherence completion of the `NARROWING-VENDING-MACHINE` module given above.

```
Maude> reduce in META-LEVEL :
metaNarrowingSearch(
upModule('NARROWING-VENDING-MACHINE, false),
'<_>['M:Money], '<_>['__['a.Item,'c.Item]], '*, unbounded, 'match, 0) .
result NarrowingSearchResult: {
  '<_>['__['a.Item,'c.Item,'#1:Money]],'State,
    'M:Money <- '__['$.Coin,'q.Coin,'q.Coin,'q.Coin,'#1:Money],
```

```
  '#,
    '#1:Money <- 'empty.Money,
  '@
}
```

Note that we obtain the very same solution, where the output contains the actual output term, its type, the accumulated substitution, the identifier used for creating fresh variables (# in this example), and the variant-unifier.

Moreover, we can also obtain the narrowing sequence associated to a narrowing-based reachability command with the function `metaNarrowingSearchPath`:

```
op metaNarrowingSearchPath :
   Module Term Term Qid Bound Qid Nat -> NarrowingSearchPathResult?
   [special ...] .
```

The result sort is defined by means of the following data:

```
sorts NarrowingSearchPathResult NarrowingSearchPathResult? .
subsort NarrowingStep < NarrowingTrace .
subsort NarrowingSearchPathResult < NarrowingSearchPathResult? .

op {_,_,_,_,_,_,_} : Context Qid Substitution Qid Term Type Substitution
               -> NarrowingStep
    [ctor format (ni n++i d ni d d d ni d ni d d d d n--i d)] .
op nil : -> NarrowingTrace [ctor] .
op __ : NarrowingTrace NarrowingTrace -> NarrowingTrace [ctor assoc id: nil] .
op {_,_,_,_,_,_} : Term Type Substitution NarrowingTrace Substitution Qid
               -> NarrowingSearchPathResult
    [ctor format (d n++i d d d d d d d d d ni n--i d)] .
op failure : -> NarrowingSearchPathResult? [ctor] .
op failureIncomplete : -> NarrowingSearchPathResult? [ctor] .
```

It works in exactly the same way as `metaNarrowingSearch` but providing as a result a more detailed data structure. If we redo the previous `metaNarrowingSearch` computation but using this time the `metaNarrowingSearchPath` function, we obtain:

```
Maude> reduce in META-LEVEL :
metaNarrowingSearchPath(
upModule('NARROWING-VENDING-MACHINE, false),
'<_>['M:Money], '<_>['__['a.Item,'c.Item]], '*, unbounded, 'none, 0) .
result NarrowingSearchPathResult: {
  '<_>['#1:Money],'State,
    'M:Money <- '#1:Money,
  {
    [],
    'buy-a,
      '#1:Money <- '__['$.Coin,'@1:Money] ;
      'M:Marking <- '@1:Money,
    '@,
    '<_>['__['a.Item,'q.Coin,'@1:Money]],'State,
      'M:Money <- '__['$.Coin,'@1:Money]
  }
  {
    [],
    'buy-c,
      '@1:Money <- '__['q.Coin,'q.Coin,'q.Coin,'#1:Money] ;
```

```
      'M:Marking <- '__['a.Item,'#1:Money],
    '#,
    '<_>['__['a.Item,'c.Item,'#1:Money]],'State,
     'M:Money <- '__['$.Coin,'q.Coin,'q.Coin,'q.Coin,'#1:Money]
  },
    '#1:Money <- 'empty.Money,
  '@
}
```

The data structure `NarrowingStep`, which is the basic element of `NarrowingStepSet`, is very similar to the data structure `ResultTriple` but it contains a sequence of narrowing results instead of only the final result, each one together with the rule that has been used in that narrowing step.

Several auxiliary functions have been defined by equations, allowing easy extraction of information.

```
op getTerm : NarrowingApplyResult -> Term .
op getType : NarrowingApplyResult -> Type .

op getContext : NarrowingApplyResult -> Context .
op getLabel : NarrowingApplyResult -> Qid .

op getTermSubstitution : NarrowingApplyResult -> Substitution .
op getRuleSubstitution : NarrowingApplyResult -> Substitution .

op getVariableFamily : NarrowingApplyResult -> Qid .

op getTerm : NarrowingSearchResult -> Term .
op getType : NarrowingSearchResult -> Type .

op getAccumulatedSubstitution : NarrowingSearchResult -> Substitution .
op getStateVariableFamily : NarrowingSearchResult -> Qid .

op getUnifier : NarrowingSearchResult -> Substitution .
op getUnifierVariableFamily : NarrowingSearchResult -> Qid .

op getInitialTerm : NarrowingSearchPathResult -> Term .
op getInitialType : NarrowingSearchPathResult -> Type .

op getInitialSubstitution : NarrowingSearchPathResult -> Substitution .
op getTrace : NarrowingSearchPathResult -> NarrowingTrace .

op getUnifier : NarrowingSearchPathResult -> Substitution .
op getUnifierVariableFamily : NarrowingSearchPathResult -> Qid .

op getContext : NarrowingStep -> Context .
op getLabel : NarrowingStep -> Qid .

op getUnifier : NarrowingStep -> Substitution .
op getUnifierVariableFamily : NarrowingStep -> Qid .

op getTerm : NarrowingStep -> Term .
op getType : NarrowingStep -> Type .
op getAccumulatedSubstitution : NarrowingStep -> Substitution .
```

17.6.12 Checking satisfiability modulo theories: metaCheck

The SMT check command of Chapter 16 is also available at the metalevel by using the function metaCheck:

```
op metaCheck : Module Term ~> Bool [special (...)] .
```

The reflection of the SMT signatures follows the normal Maude metalevel conventions, e.g., 2 becomes '2.Integer and 1/3 becomes '1/3.Real.

Consider the following example.

```
fmod META-CHECK is
  pr META-LEVEL .
  pr REAL-INTEGER .
  vars W X Y Z : Boolean .
  vars I J K L : Integer .
  vars P Q R S : Real .
endfm

Maude> red in META-LEVEL :
        metaCheck(['META-CHECK],
            upTerm( (I > J ? I : J) === I or (I > J ? I : J) === J )) .
rewrites: 3 in 6ms cpu (6ms real) (456 rewrites/second)
result Bool: (true).Bool

Maude> red in META-LEVEL :
        metaCheck(['META-CHECK],
            upTerm( not((I > J ? I : J) === I or (I > J ? I : J) === J ) )) .
rewrites: 3 in 0ms cpu (0ms real) (13824 rewrites/second)
result Bool: (false).Bool

Maude> red in META-LEVEL :
        metaCheck(['META-CHECK],
            upTerm( (I > J ? I : J) =/== I and (I > J ? I : J) =/== J )) .
rewrites: 3 in 0ms cpu (0ms real) (13698 rewrites/second)
result Bool: (false).Bool

Maude> red in META-LEVEL :
        metaCheck(['META-CHECK],
            upTerm( (I > J ? I : J) =/== I or (I > J ? I : J) =/== J )) .
rewrites: 3 in 0ms cpu (0ms real) (12931 rewrites/second)
result Bool: (true).Bool
```

Here metaCheck returns true if the SMT solver responds with sat and false otherwise; it can occasionally produce results other than sat and unsat, for example if it cannot decide satisfiability and Maude's response may change in the future.

Examples of the use of metaCheck in the analysis of time aware security protocols to reduce infinite search to a finite one can be found in [118, 119].

17.6.13 Parsing and pretty-printing: metaParse and metaPrettyPrint

metaParse

The (partial) operation metaParse takes as arguments the metarepresentation of a module, a list of quoted identifiers metarepresenting a list of tokens, and a value of the sort Type?, i.e., either the metarepresentation of a component or the constant anyType.

```
sort Type? .
subsort Type < Type? .
op anyType : -> Type? [ctor] .
sort ResultPair? .
subsort ResultPair < ResultPair? .
op noParse : Nat -> ResultPair? [ctor] .
op ambiguity : ResultPair ResultPair -> ResultPair? [ctor] .
op metaParse : Module QidList Type? ~> ResultPair? [special (...)] .
```

The operation `metaParse` reflects the `parse` command in Maude (see Section 3.9.4); that is, it tries to parse the given list of tokens as a term of the given type in the module given as first argument; the constant `anyType` allows any component. If `metaParse` succeeds, it returns the metarepresentation of the parsed term with its corresponding sort or kind. Otherwise, it returns:

- `noParse(`n`)` if there was no parse, where n is the index of the first bad token (counting from 0), or the number of tokens in the case of unexpected end of input; or
- `ambiguity(`r_1`, `r_2`)` if there were multiple parses, where r_1 and r_2 are the result pairs corresponding to two distinct parses.

These are simple examples of using `metaParse`:

```
Maude> reduce in META-LEVEL :
        metaParse(upModule('VENDING-MACHINE, false),
          '$ 'q 'q 'q, 'Marking) .
result ResultPair:
  {'__['$.Coin,'__['q.Coin,'__['q.Coin,'q.Coin]]],'Marking}

Maude> reduce in META-LEVEL :
        metaParse(upModule('VENDING-MACHINE, false),
          '$ 'q 'd 'q, 'Marking) .
result ResultPair?: noParse(2)
```

`metaPrettyPrint`

The (partial) operation `metaPrettyPrint` takes as arguments the metarepresentations of a module $\mathcal{R}$ and of a term t together with a set of printing options, and it returns a list of quoted identifiers that metarepresents the string of tokens produced by pretty-printing the term t in the signature of $\mathcal{R}$. In the event of an error an empty list of quoted identifiers is returned.

```
op metaPrettyPrint : Module Term PrintOptionSet ~> QidList
    [special (...)] .
```

Pretty-printing a term involves more than just naively using the mixfix syntax for operators. Precedence and gathering information and the relative positions of underscores in an operator and its parent in the term must be considered to determine whether parentheses need to be inserted around any given subterm to avoid ambiguity. If there is ad-hoc overloading in the module, additional checks must be done to determine if and where sort disambiguation syntax is needed.

The print options argument is built with the following syntax:

```
sorts PrintOption PrintOptionSet .
subsort PrintOption < PrintOptionSet .
ops mixfix with-parens flat format number rat : -> PrintOption
     [ctor] .
```

```
op none : -> PrintOptionSet [ctor] .
op __ : PrintOptionSet PrintOptionSet -> PrintOptionSet
        [ctor assoc comm id: none] .
```

The available print options form a subset of the global print options described in Section 23.10, which are ignored by this operation.

As an example, we can use `metaPrettyPrint` to pretty print the result of parsing at the metalevel the list of tokens `$ q q q` in the module `VENDING-MACHINE`, first with prefix syntax, then with mixfix syntax, and finally with mixfix syntax and taking into account the `format` attribute.

```
Maude> reduce in META-LEVEL :
        metaPrettyPrint(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['q.Coin, '__['q.Coin, 'q.Coin]]],
          none) .
result NeQidList:
  '__ ''( '$ '', '__ ''( 'q '', '__ ''( 'q '', 'q '') '') '')


Maude> reduce in META-LEVEL :
        metaPrettyPrint(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['q.Coin, '__['q.Coin, 'q.Coin]]],
          mixfix) .
result NeTypeList: '$ 'q 'q 'q


Maude> reduce in META-LEVEL :
        metaPrettyPrint(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['q.Coin, '__['q.Coin, 'q.Coin]]],
          mixfix format) .
result NeTypeList:
  '\r '\! '$ '\o '\r '\! 'q '\o '\r '\! 'q '\o '\r '\! 'q '\o
```

It is important to notice that `metaPrettyPrint` uses the information provided by the `format` attribute in the last reduction above. For example, the operator `$` in the module `VENDING-MACHINE-SIGNATURE` in Section 5.1 was declared with attribute `format (r! o)`, and therefore it is meta-pretty-printed as `'\r '\! '$ '\o`.

For backwards compatibility there is available the following variation of the `metaPrettyPrint` operation, which provides a set of default print options.

```
op metaPrettyPrint : Module Term ~> QidList .
eq metaPrettyPrint(M:Module, T:Term)
  = metaPrettyPrint(M:Module, T:Term,
      mixfix flat format number rat) .
```

For example,

```
Maude> reduce in META-LEVEL :
        metaPrettyPrint(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['q.Coin, '__['q.Coin, 'q.Coin]]]) .
result NeTypeList:
  '\r '\! '$ '\o '\r '\! 'q '\o '\r '\! 'q '\o '\r '\! 'q '\o
```

Parsing and pretty-printing strategy expression is also available by means of two analogous descent functions.

`metaParseStrategy`

The function `metaParseStrategy` is the counterpart of `metaParse` for strategy expressions.

```
op metaParseStrategy : Module QidSet QidList ~> Strategy? [special (...)] .
```

It takes the metarepresentation of a module along with a set of quoted identifiers representing variable declarations of the form `X:S`, and a list of quoted identifiers. It tries to parse the given tokens as a strategy expression and return its metarepresentation.

```
subsort Strategy < Strategy? .
op noStratParse : Nat -> Strategy? [ctor] .
op ambiguity  : Strategy Strategy -> Strategy? [ctor] .
```

However, if there was no parse, it returns the term `noStratParse(`n`)`, where n is the index of the first bad token (counting from 0), or the number of tokens in the case of unexpected end of input. If there are multiple parses, two of them are returned as an `ambiguity(`s_1`, `s_2`)` term. As an example, we can parse the definition of the `expand` strategy in the `QUEENS-STRAT` module:

```
Maude> red in META-LEVEL : metaParseStrategy(upModule('QUEENS-STRAT, false), none,
   'top '`( 'next '`) ';
   'match 'L:List`{Nat`} 'such 'that 'isOk '`( 'L:List`{Nat`} '`)) .
rewrites: 2 in 6ms cpu (6ms real) (301 rewrites/second)
result Strategy: top('next[none]{empty}) ;
                 match 'L:NatList s.t. ('isOk['L:NatList] = 'true.Bool)
```

Instead of writing the sort of the variable explicitly as `'L:NatList` in the input, we may pass a variable declaration for `L` to `metaParseStrategy`:

```
Maude> red metaParseStrategy(upModule('QUEENS-STRAT, false), 'L:List`{Nat`},
   'top '`( 'next '`) '; 'match 'L 'such 'that 'isOk '`( 'L '`)) .
rewrites: 2 in 3ms cpu (4ms real) (300 rewrites/second)
result Strategy: top('next[none]{empty}) ;
                 match 'L:NatList s.t. ('isOk['L:NatList] = 'true.Bool)
```

`metaPrettyPrintStrategy`

The function `metaPrettyPrintStrategy` is the counterpart of `metaPrettyPrint` for strategy expressions and the opposite of `metaParseStrategy`.

```
op metaPrettyPrintStrategy : Module Strategy PrintOptionSet
                          ~> QidList [special (...)] .
```

It takes as arguments the metarepresentation of a module and of a strategy together with a set of printing options, and returns a list of quoted identifiers that represents the string of tokens produced by pretty-printing the given strategy. The printing options are the same as for `metaPrettyPrint`. For example, the pretty-printing of the application of the rule `next` in `QUEENS-STRAT` can be obtained with:

```
Maude> red in META-LEVEL : metaPrettyPrintStrategy(
   upModule('QUEENS-STRAT, false), 'next[none]{empty}, none) .
rewrites: 2 in 46ms cpu (45ms real) (43 rewrites/second)
result Sort: 'next
```

17.6.14 Sort operations

The `META-LEVEL` module also provides in a built-in way commonly needed operations on the poset of sorts of a given module.

All these operations, related to sorts and kinds, take as first argument a term of sort `Module`. Assuming that this term is indeed the metarepresentation of a module, the remaining arguments might be terms representing sorts or kinds that do not correspond to sorts or kinds declared in such a module; in this case, the operation is undefined.

In the following we include descriptions together with simple examples of using these operations.

`sortLeq`

The operation `sortLeq` takes as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentations of two types, that is, either sorts or kinds.

```
op sortLeq : Module Type Type ~> Bool [special (...)] .
```

According to whether the types passed to `sortLeq` as arguments are metarepresented sorts or kinds, we can distinguish the following cases:

- Assume first that both types given as arguments are two sorts s and s'. Let S be the set of sorts in $\mathcal{R}$ and let $\leq_{\mathcal{R}}$ be its subsort relation. When $s, s' \in S$, `sortLeq` returns `true` if $s \leq_{\mathcal{R}} s'$ and `false` otherwise. For example,

```
Maude> reduce in META-LEVEL :
        sortLeq(upModule('NUMBERS, false), 'Zero, 'Nat) .
result Bool: true

Maude> reduce in META-LEVEL :
        sortLeq(upModule('NUMBERS, false), 'Zero, 'NzNat) .
result Bool: false
```

- If both types given as arguments are kinds in $\mathcal{R}$, then `sortLeq` returns `false` when both kinds are different and `true` when they are equal. For example,

```
Maude> reduce in META-LEVEL :
        sortLeq(upModule('NUMBERS, false), '`[Zero`], '`[Nat`]) .
result Bool: true

Maude> reduce in META-LEVEL :
       sortLeq(upModule('NUMBERS, false), '`[Zero`], '`[Bool`]) .
result Bool: false
```

- If one type is one sort in $\mathcal{R}$ and the other one is a kind in $\mathcal{R}$, then `sortLeq` checks whether the given sort belongs to the given kind or not. For example,

```
Maude> reduce in META-LEVEL :
        sortLeq(upModule('NUMBERS, false), '`[Zero`], 'Bool) .
result Bool: false

Maude> reduce in META-LEVEL :
        sortLeq(upModule('NUMBERS, false), 'Zero, '`[NatSet`]) .
result Bool: true
```

`sameKind`

The operation `sameKind` takes as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentations of two types, that is, either sorts or kinds.

```
op sameKind : Module Type Type ~> Bool [special (...)] .
```

Let S be the set of sorts in $\mathcal{R}$ and let $\leq_{\mathcal{R}}$ be its subsort relation. When the two types passed as arguments to `sameKind` are sorts $s, s' \in S$, the operation `sameKind` returns `true` if s and s' belong to the same connected component in the subsort ordering $\leq_{\mathcal{R}}$, that is, if they belong to the same kind, and `false` otherwise. When the two arguments are kinds in $\mathcal{R}$, `sameKind` returns `true` when they are indeed the same, and `false` otherwise. Finally, when one argument is one sort and the other is a kind, this operation ckecks whether the sort belongs to the kind.

For example, we have the following reductions about sorts and kinds in the module `NUMBERS`.

```
Maude> reduce in META-LEVEL :
         sameKind(upModule('NUMBERS, false), 'Zero, 'NzNat) .
result Bool: true


Maude> reduce in META-LEVEL :
         sameKind(upModule('NUMBERS, false), 'Zero, 'Nat3) .
result Bool: false


Maude> reduce in META-LEVEL :
        sameKind(upModule('NUMBERS, false), '`[Zero`], '`[NzNat`]) .
result Bool: true


Maude> reduce in META-LEVEL :
         sameKind(upModule('NUMBERS, false), '`[Zero`], 'NzNat) .
result Bool: true
```

`completeName`

The operation `completeName` takes as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentation of a sort s or a kind k. When its second argument is the metarepresentation of a sort s, it returns the same metarepresentation of s. But if its second argument is the metarepresentation of a kind k, then it returns the metarepresentation of the complete name of k in $\mathcal{R}$, i.e., the metarepresentation of the comma-separated list of the maximal elements of the corresponding connected component.

```
op completeName : Module Type ~> Type [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL :
         completeName(upModule('NUMBERS, false), 'Zero) .
result Sort: 'Zero


Maude> reduce in META-LEVEL :
         completeName(upModule('NUMBERS, false), '`[Zero`]) .
result Kind: '`[NatSeq`,NatSet`]
```

`getKind` and `getKinds`

The operation `getKind` takes as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentation of a type, i.e., a sort or a kind. When its second argument is the metarepresentation of a type in $\mathcal{R}$, it returns the metarepresentation of the complete name of the corresponding kind.

```
op getKind : Module Type ~> Kind [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL :
        getKind(upModule('NUMBERS, false), 'Zero) .
result Kind: ''[NatSeq‘,NatSet‘]

Maude> reduce in META-LEVEL :
        getKind(upModule('NUMBERS, false), ''[Zero‘]) .
result Kind: ''[NatSeq‘,NatSet‘]
```

The operation `getKinds` takes as its only argument the metarepresentation of a module $\mathcal{R}$ and returns the metarepresentation of the set of kinds declared in $\mathcal{R}$, with kinds metarepresented using their complete names.

```
op getKinds : Module ~> KindSet [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL : getKinds(upModule('NUMBERS, false)) .
result NeKindSet: ''[Bool‘] ; ''[Nat3‘] ; ''[NatSeq‘,NatSet‘]
```

`lesserSorts`

The operation `lesserSorts` takes as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentation of a type, i.e., a sort or a kind.

```
op lesserSorts : Module Type ~> SortSet [special (...)] .
```

Let S be the set of sorts in $\mathcal{R}$. When $s \in S$, `lesserSorts` returns the metarepresentation of the set of sorts strictly smaller than s in S. For example,

```
Maude> reduce in META-LEVEL :
        lesserSorts(upModule('NUMBERS, false), 'Nat) .
result NeSortSet: 'NzNat ; 'Zero

Maude> reduce in META-LEVEL :
        lesserSorts(upModule('NUMBERS, false), 'Zero) .
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
        lesserSorts(upModule('NUMBERS, false), 'NatSeq) .
result NeSortSet: 'Nat ; 'NzNat ; 'Zero
```

When the second argument of `lesserSorts` metarepresents a kind in $\mathcal{R}$, this operation returns the metarepresentation of the set of all sorts in such kind. For example,

```
Maude> reduce in META-LEVEL :
        lesserSorts(upModule('NUMBERS, false), ''[NatSeq‘]) .
result NeSortSet: 'Nat ; 'NatSeq ; 'NatSet ; 'NzNat ; 'Zero

Maude> reduce in META-LEVEL :
        lesserSorts(upModule('NUMBERS, false), ''[Bool‘]) .
result Sort: 'Bool
```

`leastSort`

The operation `leastSort` takes as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentation of a term t, and it returns the metarepresentation of the least sort or kind of t in $\mathcal{R}$, obtained without reducing the term, that is, the memberships in the module are used to get the information, but equations are not used to reduce the term.

```
op leastSort : Module Term ~> Type [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL :
   leastSort(upModule('NUMBERS, false), 'p['s_['zero.Zero]]) .
result Sort: 'Nat
```

`glbSorts`

The operation `glbSorts` takes as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentations of two types, that is, either sorts or kinds.

```
op glbSorts : Module Type Type ~> TypeSet [special (...)] .
```

According to whether the types passed to `glbSorts` as arguments are metarepresented sorts or kinds, we can distinguish the following cases:

- If both types given as arguments are sorts in $\mathcal{R}$, then `glbSorts` returns the metarepresentation of the set (which can be empty) consisting of the largest sorts that are common subsorts of the two given sorts, that is, the set of maximal lower bounds of the two sorts; when this set is a singleton set $\{s\}$, then s will be the greatest lower bound of the two sorts, thus the operation name `glbSorts`.

  For example, we have the following reductions concerning sorts in the module `NUMBERS`.

  ```
  Maude> reduce in META-LEVEL :
          glbSorts(upModule('NUMBERS, false), 'Zero, 'Nat) .
  result Sort: 'Zero

  Maude> reduce in META-LEVEL :
          glbSorts(upModule('NUMBERS, false), 'NatSet, 'NatSeq) .
  result Sort: 'Nat

  Maude> reduce in META-LEVEL :
          glbSorts(upModule('NUMBERS, false), 'NzNat, 'NzNat) .
  result Sort: 'NzNat

  Maude> reduce in META-LEVEL :
          glbSorts(upModule('NUMBERS, false), 'Zero, 'NzNat) .
  result EmptyTypeSet: (none).EmptyTypeSet

  Maude> reduce in META-LEVEL :
          glbSorts(upModule('NUMBERS, false), 'NzNat, 'Bool) .
  result EmptyTypeSet: (none).EmptyTypeSet
  ```

- If both types given as arguments are kinds in $\mathcal{R}$, then `glbSorts` returns the empty set when both kinds are different, and the metarepresentation of the kind (using the corresponding complete name) when both kinds are equal. For example,

```
Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), ''[Nat'], ''[Bool']) .
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false),''[Nat'],''[NatSeq']) .
result Kind: ''[NatSeq',NatSet']
```

- If one type is one sort in $\mathcal{R}$ and the other one is a kind in $\mathcal{R}$, then `glbSorts` returns the metarepresentation of the sort when the sort belongs to the kind, and the empty set otherwise. For example,

```
Maude> reduce in META-LEVEL :
        glbSorts(upModule('NUMBERS, false), ''[Nat'], 'Bool) .
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
        glbSorts(upModule('NUMBERS, false), ''[NatSeq'], 'Zero) .
result Sort: 'Zero

Maude> reduce in META-LEVEL :
       glbSorts(upModule('NUMBERS, false), 'NzNat, ''[NatSet']) .
result Sort: 'NzNat
```

`maximalSorts` **and** `minimalSorts`

The operations `maximalSorts` and `minimalSorts` take as arguments the metarepresentation of a module $\mathcal{R}$ and the metarepresentation of a kind k. If k is a kind in $\mathcal{R}$, `maximalSorts` returns the metarepresentation of the set of the maximal sorts in the connected component of k, while `minimalSorts` returns the metarepresentation of the set of its minimal sorts.

```
op maximalSorts : Module Kind ~> SortSet [special (...)] .
op minimalSorts : Module Kind ~> SortSet [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL :
   maximalSorts(upModule('NUMBERS, false), ''[Zero']) .
result NeSortSet: 'NatSeq ; 'NatSet

Maude> reduce in META-LEVEL :
   minimalSorts(upModule('NUMBERS, false), ''[Zero']) .
result NeSortSet: 'Zero ; 'NzNat
```

`maximalAritySet`

The operation `maximalAritySet` takes as arguments the metarepresentation of a module $\mathcal{R}$, the metarepresentation of an operator f in $\mathcal{R}$, the metarepresentation of an arity (list of types) for f and the metarepresentation of a sort s, and then computes the set of maximal arities (lists of types) that f could take and have a sort $s' \leq_{\mathcal{R}} s$. This result might be the empty set if s is small or f is only defined at the kind level.

Notice that the result of this operation is a *set of lists* of types, which is built by means of the following syntax, extending the syntax for building lists of types that we only show partially here and whose full specification can be found in the module `META-MODULE` in the file `prelude.maude` available with the Maude distribution.

```
sort NeTypeList TypeList .
op nil : -> TypeList [ctor] .
op __ : TypeList TypeList -> TypeList [ctor ditto] .

sort TypeListSet .
subsort TypeList TypeSet < TypeListSet .
op _;_ : TypeListSet TypeListSet -> TypeListSet [ctor ditto] .
eq T:TypeList ; T:TypeList = T:TypeList .

op maximalAritySet : Module Qid TypeList Sort ~> TypeListSet
  [special (...)] .
```

Let us consider for example the operator `_+_` in the module `NUMBERS`, where it is overloaded by means of the following declarations:

```
op _+_ : Nat Nat -> Nat [assoc comm].
op _+_ : NzNat Nat -> NzNat [ditto] .
op _+_ : Nat3 Nat3 -> Nat3 [comm] .
```

With this information, we obtain the following reductions concerning this operator:

```
Maude> reduce in META-LEVEL :
        maximalAritySet(upModule('NUMBERS, false),
          '_+_, 'NzNat 'NzNat, 'NzNat) .
result TypeListSet: 'Nat 'NzNat ; 'NzNat 'Nat


Maude> reduce in META-LEVEL :
        maximalAritySet(upModule('NUMBERS, false),
          '_+_, 'Nat 'Nat, 'NzNat) .
result TypeListSet: 'Nat 'NzNat ; 'NzNat 'Nat


Maude> reduce in META-LEVEL :
        maximalAritySet(upModule('NUMBERS, false),
          '_+_, 'Nat 'Nat, 'Nat) .
result NeTypeList: 'Nat 'Nat


Maude> reduce in META-LEVEL :
        maximalAritySet(upModule('NUMBERS, false),
          '_+_, 'Nat3 'Nat3, 'Nat3) .
result NeTypeList: 'Nat3 'Nat3
```

Notice that if the operator f and the list of types passed as arguments to `maximalAritySet` do not match, then the result is an error, which is represented as a non-reduced term in a metalevel kind. We have for instance the following example where we have omitted the lengthy metarepresentation of the `NUMBERS` module.

```
Maude> reduce in META-LEVEL :
        maximalAritySet(upModule('NUMBERS, false),
          '_+_, 'Nat3 'Nat3, 'NzNat) .
result [GTermList,ParameterList,QidList,
       TypeListSet,Type?,ModuleExpression,Header]:
  maximalAritySet(fmod 'NUMBERS is ... endfm,
    '_+_, 'Nat3 'Nat3, 'NzNat)
```

17.6.15 Other metalevel operations: `wellFormed`

The operation `wellFormed` can take as arguments the metarepresentation of a module $\mathcal{R}$, or the metarepresentation of a module $\mathcal{R}$ and a term t, or the metarepresentation of a module $\mathcal{R}$ and a substitution σ. In the first case, it returns `true` if $\mathcal{R}$ is a well-formed module, and `false` otherwise. In the second case, if t is a well-formed term in $\mathcal{R}$, it returns `true`; otherwise, it returns `false`. Finally, in the third case, if σ is a well-formed substitution in $\mathcal{R}$, it returns `true`; otherwise, it returns `false`.

```
op wellFormed : Module -> Bool [special (...)] .
op wellFormed : Module Term ~> Bool [special (...)] .
op wellFormed : Module Substitution ~> Bool [special (...)] .
```

Note that the first operation is total, while the other two are partial (notice the form of the arrow in the declarations). The reason is that the last two are not defined when the term given as first argument does not represent a module, and then it does not make sense to check whether a term or substitution is well formed with respect to such a wrong "module." For example,

```
Maude> reduce in META-LEVEL :
        wellFormed(upModule('NUMBERS, false)) .
result Bool: true

Maude> reduce in META-LEVEL :
        wellFormed(upModule('NUMBERS, false), 'p['zero.Zero]) .
result Bool: true

Maude> reduce in META-LEVEL :
        wellFormed(upModule('NUMBERS, false),
          's_['zero.Zero, 'zero.Zero]) .
Advisory: could not find an operator s_ with appropriate domain
    in meta-module NUMBERS when trying to interprete metaterm
    's_['zero.Zero,'zero.Zero].
result Bool: false

Maude> reduce in META-LEVEL :
        wellFormed(upModule('NUMBERS, false),
          'N:Zero <- 'zero.Zero) .
result Bool: true

Maude> reduce in META-LEVEL :
        wellFormed(upModule('NUMBERS, false),
          'N:Nat <- 'p['zero.Zero]) .
result Bool: false

Maude> reduce in META-LEVEL :
        wellFormed(upModule('NUMBERS, false),
          'N:Zero <- 's_['zero.Zero,'zero.Zero]) .
Advisory: could not find an operator s_ with appropriate domain
    in meta-module NUMBERS when trying to interprete metaterm
    's_['zero.Zero,'zero.Zero].
result Bool: false
```

17.7 Internal strategies

System modules in Maude are rewrite theories that do not need to be Church-Rosser and terminating. Therefore, we need to have good ways of controlling the rewriting inference process—which in principle could not terminate or go in many undesired directions—by means of adequate *strategies*, as already explained in Chapter 10.

In Maude, thanks to its reflective capabilities, strategies can be made *internal* to the system. That is, they can be defined using statements in a normal module in Maude, and can be reasoned about as with statements in any other module. In general, strategies are defined in extensions of the `META-LEVEL` module by using `metaReduce`, `metaApply`, `metaXapply`, etc., as building blocks.

We illustrate some of these possibilities by implementing the following strategies for controlling the execution of the rules in the `VENDING-MACHINE` module in Section 5.1:

1. insert either a dollar or a quarter in the vending machine;
2. only buy cakes, and buy as many cakes as possible, with the coins already inserted;
3. only buy either cakes or apples, and buy at most n of them, with the coins already inserted;
4. buy the same number of apples and cakes, and buy as many as possible, with the coins already inserted.

Consider the module `BUYING-STRATS` below, which imports the `META-LEVEL` module.

```
fmod BUYING-STRATS is
  protecting META-LEVEL .
```

The function `insertCoin` below defines the strategy (1): it expects as first argument either `'add-q` or `'add-$`, for inserting a quarter or a dollar, respectively, and as second argument the metarepresentation of the marking of a vending machine, and it applies once the rule corresponding to the given label. The rules `add-q` and `add-$` are applied using the descent function `metaXapply`. A rule cannot be applied when the result of `metaXapply`-ing the rule is not a term of sort `Result4Tuple`. Note the use of a matching equation in the condition to simplify the presentation of the righthand side of the equation (see Section 4.3), as well as the use of the statement attribute `owise` (see Section 4.5.4) to define the function `insertCoin` for unexpected cases.

```
  var T : Term .
  var Q : Qid .
  var  N : Nat .
  vars BuyItem? BuyCake? Change? : [Result4Tuple].

  op insertCoin : Qid Term -> Term .

  ceq insertCoin(Q, T)
    = if BuyItem? :: Result4Tuple
      then getTerm(BuyItem?)
      else T
      fi
    if (Q == 'add-q or Q == 'add-$)
       /\ BuyItem? := metaXapply(upModule('VENDING-MACHINE, false),
                      T, Q, none, 0, unbounded, 0) .

  eq insertCoin(Q, T) = T [owise] .
```

The function `onlyCakes` below defines the strategy (2): it applies the rule `buy-c` as many times as possible, applying the rule `change` whenever it is necessary. In particular, if the rule `buy-c` can be applied, then there is a recursive call to the function `onlyCakes` with the term resulting from its application. If the rule `buy-c` cannot be applied, then the application of the rule `change` is attempted. If the rule `change` can be applied, then there is a recursive call to the function `onlyCakes` with the term resulting from the `change` rule application. Otherwise, the argument is returned unchanged. The rules `buy-c` and `change` are also applied using the descent function `metaXapply`.

```
op onlyCakes : Term -> Term .
ceq onlyCakes(T)
  = if BuyCake? :: Result4Tuple
    then onlyCakes(getTerm(BuyCake?))
    else (if Change? :: Result4Tuple
          then onlyCakes(getTerm(Change?))
          else T
          fi)
    fi
  if BuyCake? := metaXapply(upModule('VENDING-MACHINE, false),
                 T, 'buy-c, none, 0, unbounded, 0)
     /\ Change? := metaXapply(upModule('VENDING-MACHINE, false),
                   T, 'change, none, 0, unbounded, 0) .
```

The function `onlyNitems` defines the strategy (3): it applies either the rule `buy-c` or `buy-a` (but not both) at most n times. As expected, the rules are applied using the descent function `metaXapply`. Note the use of the symmetric difference operator `sd` (see Section 7.2) to decrement `N`.

```
op onlyNitems : Term Qid Nat -> Term .

ceq onlyNitems(T, Q, N)
  = if N == 0
    then T
    else (if BuyItem? :: Result4Tuple
          then onlyNitems(getTerm(BuyItem?), Q, sd(N, 1))
          else (if Change? :: Result4Tuple
                then onlyNitems(getTerm(Change?), Q, N)
                else T
                fi)
          fi)
    fi
  if (Q == 'buy-c or Q == 'buy-a)
     /\ BuyItem? := metaXapply(upModule('VENDING-MACHINE, false),
                    T, Q, none, 0, unbounded, 0)
     /\ Change? := metaXapply(upModule('VENDING-MACHINE, false),
                   T, 'change, none, 0, unbounded, 0) .

eq onlyNitems(T, Q, N) = T [owise] .
```

Finally, the function `cakesAndApples` defines the strategy (4): it applies the rule `buy-c` as many times as the rule `buy-a`. To define this function, we use an auxiliary Boolean function `buyItem?` that determines whether a given rule (`buy-c` or `buy-a`) can be applied. In the definition of `cakesAndApples` the Boolean function `buyItem?` is used to check if the rule `buy-a` can be applied after applying the rule `buy-c`. When the answer is `true`, then `buy-c` and

buy-a are applied once, using the function onlyNitems with the appropriate arguments, and the function cakesAndApples is applied again to the result.

```
op cakesAndApples : Term -> Term .
op buyItem? : Term Qid -> Bool .

ceq buyItem?(T, Q)
  = if BuyItem? :: Result4Tuple
    then true
    else (if Change? :: Result4Tuple
          then buyItem?(getTerm(Change?), Q)
          else false
          fi)
    fi
  if (Q == 'buy-c or Q == 'buy-a)
     /\ BuyItem? := metaXapply(upModule('VENDING-MACHINE, false),
                               T, Q, none, 0, unbounded, 0)
     /\ Change? := metaXapply(upModule('VENDING-MACHINE, false),
                              T, 'change, none, 0, unbounded, 0) .

eq buyItem?(T, Q) = false [owise] .

eq cakesAndApples(T)
  = if buyItem?(T, 'buy-a)
    then (if buyItem?(onlyNitems(T, 'buy-a, 1), 'buy-c)
          then cakesAndApples(onlyNitems(onlyNitems(T, 'buy-a, 1),
                                         'buy-c, 1))
          else T
          fi)
    else T
    fi .
endfm
```

As examples, we apply below the buying strategies (2–4) to spend in different ways the same amount of money: three dollars and a quarter.

```
Maude> reduce in BUYING-STRATS :
        onlyCakes('__['$.Coin, '$.Coin, '$.Coin, 'q.Coin]) .
result GroundTerm: '__['q.Coin, 'c.Item, 'c.Item, 'c.Item]

Maude> reduce in BUYING-STRATS :
        onlyNitems('__['$.Coin, '$.Coin, '$.Coin, 'q.Coin],
          'buy-a, 3) .
result GroundTerm:
  '__['q.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'a.Item, 'a.Item, 'a.Item]

Maude> reduce in BUYING-STRATS :
        cakesAndApples('__['$.Coin, '$.Coin, '$.Coin, 'q.Coin]) .
result GroundTerm: '__['$.Coin, 'q.Coin, 'q.Coin, 'a.Item, 'c.Item]
```

There is in fact great freedom for defining many different types of strategies, or even many different strategy languages inside Maude. As illustrated above with simple examples, this can be done in a completely user-definable way, so that users are not limited by a fixed and closed particular strategy language. Another example is presented in Section 22.6. See [25] for a general methodology for defining internal strategy languages using reflection, and [26, 28] for other examples of rewriting strategies defined in Maude.

However, the great freedom of defining internal strategies at the metalevel is purchased at some cost. First, some familiarity with Maude's metalevel features is required; and second, some cost in performance is incurred in comparison with what might be possible in a direct implementation using Maude's rewrite engine. To address these two issues, a strategy language for Maude, that can be used entirely at the object level, has been designed and implemented, as described in Chapter 10. For example, the strategy for buying the same number of apples and cakes can be written in this language as

```
((buy-a or-else change ; buy-a) ; (buy-c or-else change ; buy-c)) !
```

Chapter 18

User Interfaces and Metalanguage Applications

This chapter explains how to use the facilities provided by the predefined modules `META-LEVEL`, `STD-STREAM`, and `LEXICAL` for constructing user interfaces and metalanguage applications, in which Maude is used not only to define a domain-specific language or tool, but also to build an environment for the given language or tool. In such applications, the `STD-STREAM` module can be used to handle the input/output and to maintain the persistent state of the language environment or tool.[1]

18.1 User interfaces

In order to generate in Maude an *interface* for an application $\mathcal{P}$, the first thing we need to do is to define the language for interaction. This can be done by defining the datatype of its commands and other constructs by means of a signature $Sign_{\mathcal{P}}$.

As a running example for this chapter, we will specify a basic interface for the vending machine introduced in Section 5.1. First, we define in the module `VENDING-MACHINE-GRAMMAR` a language for interacting with the vending machine. The signature of this module extends the signature of `VENDING-MACHINE-SIGNATURE` with operators to represent the valid actions, namely: `insert $` and `insert q` for inserting a dollar or a quarter in the machine; `show basket` and `show credit` for showing the items already bought or the remaining credit; and `buy__(s)` for buying a number of pieces of the same item.

```
fmod VENDING-MACHINE-GRAMMAR is
  protecting VENDING-MACHINE-SIGNATURE .
  protecting NAT .
  sort Action .
  op insert $ : -> Action .
  op insert q : -> Action .
  op show basket : -> Action .
  op show credit : -> Action .
  op buy__(s) : Nat Item -> Action .
endfm
```

[1] Maude's prelude also includes the module `LOOP-MODE`, which was the main tool to create interactive applications in previous releases of Maude. The functionality in `STD-STREAM` is more general and flexible, and the `LOOP-MODE` has been deprecated. It is included in the current release for backwards compatibility.

The interaction with the standard IO streams is performed using objects and messages. The next step is therefore to define the objects that will keep the state of the system and will interact with the IO streams. Our goal is to have a loop between the system and the user in which the vending machine asks for input, and then, depending on the specific action, performs one task or another, and gives its response to the user.

In the module `VENDING-MACHINE-IO` below, we introduce the necessary declarations to use objects of a class `VM` with attributes `action`, to keep the last action requested from the user (inserting a coin, showing information about the remaining credit or the items already bought, or buying one or more items), and `marking`, to keep the state of the machine (the *marking* of the vending machine, that is, the remaining credit plus the items already bought). The constants `vm` and `idle` are introduced for convenience to denote the identifier of the vending machine object and to denote the idle action in which the system is waiting for input from the user once the actual action is processed.

```
mod VENDING-MACHINE-IO is
  including STD-STREAM .
  including VENDING-MACHINE-GRAMMAR .
  protecting BUYING-STRATS .
  protecting CONVERSION .
  protecting LEXICAL .

  sort VM .
  subsort VM < Cid .
  op VM : -> VM .
  op action :_ : Action -> Attribute .
  op marking :_ : Marking -> Attribute .

  op vm : -> Oid .
  op idle : -> Action .
```

To operate on the system we introduce two further constants: `VM-GRAMMAR` represents the module in which the input from the user is going to be parsed, the above `VENDING-MACHINE-GRAMMAR`; the `vending machine` configuration is used to initialize the system, and it includes the portal to interact with standard streams, the `vm` object, initially with `null` marking and action `idle`, and a first `write` message to show the user that the machine is started.

```
  op VM-GRAMMAR : -> Module .
  eq VM-GRAMMAR = upModule('VENDING-MACHINE-GRAMMAR, false) .

  op vending machine : -> Configuration .
  eq vending machine
   = <>
     < vm : VM | marking : null, action : idle >
     write(stdout, vm, "\n\t Vending machine\n") .
```

The interaction with the user is then handled by the following two rules. Once a `write` message is processed, the output stream sends back a `wrote` message, which is used to request an action from the user, who is prompted with a `>` character. When the user hits the return key, the stream object sends the input in a `gotLine` message. In the rule below, we expect either a `"quit"` that will finish the interaction, a valid input in the `VM-GRAMMAR` grammar, or a non-valid input. In the first case, a goodbye message is shown and the interaction terminates (note that the object is removed from the state). If there is an invalid input, an error message is shown and the interaction continues. If the input is valid in the `VM-GRAMMAR` grammar, the corresponding action is placed in the `action` attribute of the `VM` object for further processing.

Note the use of the `tokenize` function[2] to tokenize the input, which is received as a string, before parsing it. The operation `metaParse` checks whether the input stream corresponds to a term of sort `Action`. If this is the case, `metaParse` returns the metarepresentation of that term, which is then "moved down" using the `META-LEVEL` function `downTerm` (see Section 17.6.1).

```
vars O O' : Oid .
var  Str : String .
var  Atts : AttributeSet .
var  X : VM .

rl < O : X | Atts >
   wrote(O, O')
=> < O : X | Atts >
   getLine(stdin, O, "> ") .

rl < O : X | action : idle, Atts >
   gotLine(O, O', Str)
=> if Str == "quit"
   then write(stdout, O, "goodbye\n")
   else if metaParse(VM-GRAMMAR, tokenize(Str), 'Action) :: ResultPair
        then < O : X |
               action : downTerm(
                          getTerm(
                            metaParse(VM-GRAMMAR, tokenize(Str), 'Action)),
                          idle),
               Atts >
        else < O : X | action : idle, Atts >
             write(stdout, O, "Invalid input\n")
        fi
   fi .
```

The processing of the valid inputs, the actions, is performed by several rewriting rules, modeling the changes produced in the state of the vending machine by the actions requested by the client. To define the interaction of the state of the vending machine with the client, we can use the strategies introduced in the `BUYING-STRATS` module described in Section 17.7. Recall that `BUYING-STRATS` includes the `META-LEVEL` module.

```
var  A : Action .
var  I : Item .
var  C : Coin .
var  M : Marking .
vars QIL QIL' QIL'' : QidList .
var  N : Nat .
```

For the `show basket` and `show credit` actions, the following rules extract the information about the remaining credit or the items already bought, and send a `write` message with the corresponding information. In the definitions of the auxiliary functions `showBasket` and `showCredit`, the operation `metaPrettyPrint` takes the metarepresentation of a coin or an item, and returns the list of quoted identifiers that encode the list of tokens produced by pretty-printing the coin or the item in the module `VENDING-MACHINE-SIGNATURE`. Coins and items, and, more generally, markings of a vending machine are metarepresented using the `META-LEVEL` function `upTerm` (see Section 17.6.1).

[2]The `tokenize` function tokenizes strings into lists of quoted identifiers (see Section 7.11).

```
op showBasket : Marking -> QidList .
eq showBasket(I M)
   = metaPrettyPrint(upModule('VENDING-MACHINE-SIGNATURE, false), upTerm(I))
     showBasket(M) .
eq showBasket(C M) = showBasket(M) .
eq showBasket(null) = nil .

op showCredit : Marking -> QidList .
eq showCredit(C M)
   = metaPrettyPrint(upModule('VENDING-MACHINE-SIGNATURE, false), upTerm(C))
     showCredit(M) .
eq showCredit(I M) = showCredit(M) .
eq showCredit(null) = nil .

rl < O : X | action : show basket, marking : M, Atts >
=> < O : X | action : idle, marking : M, Atts >
   write(stdout, O, "basket: " + printTokens(showBasket(M)) + "\n") .

rl < O : X | action : show credit, marking : M, Atts >
=> < O : X | action : idle, marking : M, Atts >
   write(stdout, O, "credit: " + printTokens(showCredit(M)) + "\n") .
```

The following rules implement the actions of inserting a dollar or a quarter in the vending machine. The strategy `insertCoin` defined in the module `BUYING-STRATS` (see Section 17.7) is used to produce the corresponding change in the current marking of the vending machine. Since strategies are applied at the metalevel, both the marking of the vending machine and the coin to be inserted must be first metarepresented using again the `META-LEVEL` function `upTerm`.

```
rl < O : X | action : insert q, marking : M, Atts >
=> < O : X | action : idle,
      marking : downTerm(insertCoin('add-q, upTerm(M)), null), Atts >
   write(stdout, O, "one quarter introduced\n") .

rl < O : X | action : insert $, marking : M, Atts >
=> < O : X | action : idle,
     marking : downTerm(insertCoin('add-$, upTerm(M)), null), Atts >
   write(stdout, O, "one dollar introduced\n") .
```

The last two rules implement the actions of buying one or more items. The strategy `onlyNitems` defined in the module `BUYING-STRATS` (see Section 17.7) is used to produce the corresponding change in the current marking of the vending machine. Again, since strategies are applied at the metalevel, the marking of the vending machine must be first metarepresented.

```
rl < O : X | action : (buy N c (s)), marking : M, Atts >
=> < O : X | action : idle,
      marking : downTerm(onlyNitems(upTerm(M), 'buy-c, N), null), Atts >
   write(stdout, O, string(N, 10) + " cakes bought\n") .

rl < O : X | action : (buy N a (s)), marking : M, Atts >
=> < O : X | action : idle,
     marking : downTerm(onlyNitems(upTerm(M), 'buy-a, N), null),Atts >
  write(stdout, O, string(N, 10) + " apples bought\n") .
endm
```

18.2 The interaction with the system

With the above definitions, we can now illustrate the basic interaction with the vending machine. Once the `VENDING-MACHINE-IO` module has been entered, we can initiate the execution using the `erewrite` command and the `vending machine` constant.

```
Maude> erew vending machine .
erewrite in VENDING-MACHINE-IO : vending machine .

        Vending machine
>
```

Once the machine interface has been initialized, we can input any data by writing it after the prompt. For example,

```
> insert $
one dollar introduced

> show credit
credit: $

> insert $
one dollar introduced

> insert q
one quarter introduced

> buy 1 apple(s)
Invalid input

> buy 1 a(s)
1 apples bought

> show basket
basket: a

> show credit
credit: $ q q

> insert $
one dollar introduced

> buy 3 a(s)
3 apples bought

> show basket
basket: a a a a

> show credit
credit: q
```

We can terminate the interface by introducing the `quit` action.

```
> quit
goodbye
rewrites: 428 in 13ms cpu (83975ms real) (32552 rewrites/second)
result Portal: <>
```

18.3 Metalanguage applications: tokens, bubbles, and metaparsing

The example presented in the previous two sections is a toy example to illustrate the basic features of a possible interaction loop with the system. However, the most interesting applications of the `STD-STREAM` module are *metalanguage* applications, in which Maude is used to define the syntax, parse, execute, and pretty print the execution results of a given object language or tool. In such applications, most of the hard work is done by the `META-LEVEL` module, but we can use a similar approach to handle the input/output and maintaining the persistent state of the object language interpreter or tool.

In order to generate in Maude an *environment* for a language $\mathcal{L}$, including the case of a language with user-definable syntax, the first thing we need to do is to define the syntax for $\mathcal{L}$-modules. This can be done by defining a datatype for $\mathcal{L}$-modules, as well as auxiliary declarations for commands and other constructs, by means of a signature $Sign_{\mathcal{L}}$. Maude provides great flexibility to do this, thanks to its mixfix front-end and to the use of *bubbles* (any non-empty list of Maude identifiers). The intuition behind bubbles is that they correspond to pieces of a module in a language that can only be parsed once the grammar introduced by the signature of the module is available.

The idea is that, for a language that allows modules with user-definable syntax—as it is the case for Maude itself—it is natural to see its syntax as a combined syntax at two different levels: (1) what we may call the top-level syntax of the language, and (2) the user-definable syntax introduced in each module. The bubble datatype allows us to reflect this duality of levels in the syntax definition by encapsulating portions of (as yet unparsed) text in the user-definable syntax. Similar ideas have been exploited using ASF+SDF [41, 42].

To illustrate this concept, suppose that we want to define the syntax of Maude in Maude. Consider the following Maude module:

```
fmod NAT3 is
  sort Nat3 .
  op s_ : Nat3 -> Nat3 .
  op 0 : -> Nat3 .
  eq [ s s s 0 ] = [ 0 ] .
endfm
```

Notice that the lists of characters inside the boxes are not part of the top level syntax of Maude and therefore should be treated as bubbles until they are parsed. In fact, they can only be parsed with the grammar associated with the signature of the module `NAT3`. In this sense, we say that the syntax for Maude modules is a combination of two levels of syntax. The term `s s s 0`, for example, has to be parsed in the grammar associated with the signature of `NAT3`. The definition of the syntax of Maude in Maude must reflect this duality of syntax levels.

So far, we have talked about bubbles in a generic way. In fact, there can be many different kinds of bubbles. In Maude we can define different types of bubbles as built-in datatypes by parameterizing their definition. Thus, for example, a bubble of length one, which we call a *token*, can be defined as follows:

```
sort Token .
op token : Qid -> Token
    [special (id-hook Bubble    (1 1)
              op-hook qidSymbol (<Qids> : ~> Qid))] .
```

Any name can be used to define a bubble sort. It is the `special` attribute

```
id-hook Bubble     (1 1)
```

in its constructor declaration that makes the sort `Token` a bubble sort. The second argument of the `id-hook` special attribute indicates the minimum and maximum length of such bubbles as lists of identifiers. Therefore, `Token` has only bubbles of size 1. To specify a bubble of any length we would use the pair of values `1` and `-1`. The operator used in the declaration of the bubble, in this case the operator `token`, is a bubble constructor that represents tokens in terms of their quoted form. For example, the token `abc123` is represented as `token('abc123)`.

We can define bubbles of any length, that is, non-empty sequences of Maude identifiers, with the following declarations.

```
op bubble : QidList -> Bubble
     [special
        (id-hook Bubble      (1 -1)
         op-hook qidListSymbol (__ : QidList QidList ~> QidList)
         op-hook qidSymbol   (<Qids> : ~> Qid)
         id-hook Exclude     (.))] .
```

In this case, the system will represent the bubble as a list of quoted identifiers under the constructor `bubble`. For example, the bubble `ab cd ef` is represented as `bubble('ab 'cd 'ef)`.

Different types of bubbles can be defined using the `id-hook` special attribute `Exclude`, which takes as parameter a list of identifiers to be excluded from the given bubble, that is, the bubble being defined cannot contain such identifiers. In general, the syntax `Exclude (I1 I2 ... Ik)` is used to exclude identifiers $\texttt{I}_1, \texttt{I}_2, \ldots, \texttt{I}_k$ inside tokens.

We can, for example, declare the sort `NeTokenList` with constructor `neTokenList` as a list of identifiers, of any length greater or equal than one, excluding the identifier '`->`' with the following declarations.

```
op neTokenList : QidList -> NeTokenList
     [special
        (id-hook Bubble      (1 -1)
         op-hook qidListSymbol (__ : QidList QidList ~> QidList)
         op-hook qidSymbol   (<Qids> : ~> Qid)
         id-hook Exclude     ( -> ))] .
```

We are now ready to give the signature to parse modules such as `NAT3` above. The following module `MINI-MAUDE-SYNTAX` uses the above definitions of sorts `Token`, `Bubble` and `NeTokenList` to define the syntax of a sublanguage of Maude, namely, many-sorted, unconditional, functional modules, in which the declarations of sorts and operators have to be done one at a time, no attributes are supported for operators, and variables must be declared on-the-fly.

```
fmod MINI-MAUDE-SYNTAX is
  protecting QID-LIST .
  sorts Token Bubble NeTokenList .
  op token : Qid -> Token
       [special
          (id-hook Bubble      (1 1)
           op-hook qidSymbol   (<Qids> : ~> Qid))] .

  op bubble : QidList -> Bubble
       [special
          (id-hook Bubble      (1 -1)
           op-hook qidListSymbol (__ : QidList QidList ~> QidList)
           op-hook qidSymbol   (<Qids> : ~> Qid)
           id-hook Exclude     (.))] .

  op neTokenList : QidList -> NeTokenList
```

```
      [special
         (id-hook Bubble      (1 -1)
          op-hook qidListSymbol (__ : QidList QidList ~> QidList)
          op-hook qidSymbol   (<Qids> : ~> Qid)
          id-hook Exclude     (->))] .

  sorts Decl DeclList PreModule Input Command .
  subsort Decl < DeclList .
  subsorts Module Command < Input .

  op including_. : Token -> Decl .     --- including declaration
  op sort_. : Token -> Decl .          --- sort declaration
  op op_: ->_. : Token Token -> Decl . --- operator declaration
  op op_:_->_. : Token NeTokenList Token -> Decl .
  op eq_=_. : Bubble Bubble -> Decl .  --- equation declaration

  --- functional module
  op fmod_is_endfm : Token DeclList -> PreModule .
  op __ : DeclList DeclList -> DeclList [assoc gather(e E)] .

  --- reduce command
  op reduce_. : Bubble -> Command .
endfm
```

Notice how we explicitly declare operators that correspond to part of the top-level syntax of Maude, and how we represent as terms of sort `Bubble` those pieces of the module—namely, terms in equations—that can only be parsed afterwards with the user-defined syntax. Notice also that not all terms of sort `PreModule` represent valid Maude modules. In particular, for a term of sort `PreModule` to represent a Maude module all the bubbles must be correctly parsed as terms in the module's user-defined syntax. We sometimes refer to modules with bubbles as *premodules*.

As an example, we can call the operation `metaParse`, from module `META-LEVEL`, with the metarepresentation of the module `MINI-MAUDE-SYNTAX` and the previous module `NAT3` transformed into a list of quoted identifiers.

```
Maude> red in META-LEVEL :
          metaParse(upModule('MINI-MAUDE-SYNTAX, false),
            'fmod 'NAT3 'is
               'sort 'Nat3 '.
               'op 's_ ': 'Nat3 '-> 'Nat3 '.
               'op '0 ': '-> 'Nat3 '.
               'eq 's 's 's '0 '= '0 '.
            'endfm,
            'PreModule) .
```

We get the following term of sort `ResultPair` as a result:

```
result ResultPair:
  {'fmod_is_endfm['NAT3],
     '__['sort_.['Nat3],
     '__['op_:_->_.['s_], ['Nat3], ['Nat3]],
     '__['op_:_->_.['0], ['Nat3]],
     'eq_=_.['s 's 's '0], ['0]]],
   'PreModule}
```

Of course, Maude does not return these boxes. Instead, the system returns the bubbles using their constructor form as specified in their corresponding declarations. For example, the bubbles `'Nat3` and `'s 's 's '0` are represented, respectively, as `token('Nat3)` and `bubble('s 's 's '0)`. Maude returns them metarepresented. The result given by Maude is therefore the following.

```
result ResultPair: {
  'fmod_is_endfm['token[''NAT3.Qid],
    '__['sort_.['token[''Nat3.Qid]],
    '__['op_:_->_.['token[''s_.Qid],
                   'neTokenList[''Nat3.Qid],
                   'token[''Nat3.Qid]],
    '__['op_:'->_.['token[''0.Qid], 'token[''Nat3.Qid]],
    'eq_=_.['bubble['__[''s.Qid, ''s.Qid, ''s.Qid,''0.Qid]],
           'bubble[''0.Qid]]]]],
  'PreModule}
```

The first component of the result pair is a metaterm of sort `Term`. To convert this term into a term of sort `FModule` is now straightforward. As already mentioned, we first have to extract from the term the module's signature. For this, we can use an equationally defined function

```
op extractSignature : Term ~> FModule .
op extractSignature : Term FModule ~> FModule .
```

that goes along the term metarepresenting the premodule looking for sort, operator, and importation declarations. A homonymous function is used to recursively handle each declaration and add it to the module carried along as second argument. Notice that the operation `extractSignature` is partial, because it is not well defined for metaterms of sort `Term` that do not metarepresent terms of sort `PreModule` in `MINI-MAUDE-SYNTAX`.

Once we have extracted the signature of the module—expressed as a functional module with no equations and no membership axioms—we can then build terms of sort `EquationSet` with an equationally defined operation `solveBubbles` (also partial) that recursively replaces each bubble in an equation with the result of calling `metaParse` with the already extracted signature and with the quoted identifier form of the bubble.

```
op solveBubbles : Term FModule ~> FModule .
op solveBubblesAux : Term FModule ~> EquationSet .
```

Finally, the partial operation `processModule` takes a term and, if it metarepresents a term of sort `PreModule` in `MINI-MAUDE-SYNTAX`, and, furthermore, the `solveBubbles` function succeeds in parsing the bubbles in equations as terms, then it returns a term of sort `FModule`.

The complete specification of these operations is as follows:

```
fmod MINI-MAUDE is
  protecting META-LEVEL .

  vars T T1 T2 T3 : Term .
  vars TL TL' : TermList .
  var  QI : Qid .
  var  QIL : QidList .
  var  F : Qid .
  var  M : Module .
  var  I : Import .
  vars IL : ImportList .
  var  S : Sort .
```

```
vars SS : SortSet .
var  SsS : SubsortDeclSet .
var  OD : OpDecl .
var  ODS : OpDeclSet .
var  MbS : MembAxSet .
vars EqS EqS' : EquationSet .

op processModule : Term ~> FModule .
eq processModule(T) = solveBubbles(T, extractSignature(T)) .

---- extractSignature
op extractSignature : Term ~> FModule .
op extractSignature : Term FModule ~> FModule .
eq extractSignature('fmod_is_endfm['token[QI], T])
 = extractSignature(T,
    fmod downTerm(QI, 'error) is nil sorts none . none none none none endfm) .

eq extractSignature('__[T1, T2], M)
 = extractSignature(T2, extractSignature(T1, M)) .
eq extractSignature('sort_.['token[T]], M) = addSort(M, downTerm(T, 'error)) .
eq extractSignature('op_:_->_.['token[T1], 'neTokenList[TL], 'token[T2]], M)
 = addOpDecl(M,
    op downTerm(T1, 'error) : downTerm(TL, nil) -> downTerm(T2, 'error) [none] .) .
eq extractSignature('op_:`->_.['token[T1], 'token[T2]], M)
 = addOpDecl(M, (op downTerm(T1, 'error) : nil -> downTerm(T2, 'error) [none] .)) .
eq extractSignature('including_.['token[T]], M)
 = addImport(M, including downTerm(T, 'error) .) .
eq extractSignature(T, M) = M [owise] .

---- solveBubbles
op solveBubbles : Term FModule ~> FModule .
op solveBubblesAux : Term FModule ~> EquationSet .

eq solveBubbles('fmod_is_endfm['token[QI], T], M)
  = addEquations(M, solveBubblesAux(T, M)) .

eq solveBubblesAux('eq_=_.[T1, T2], M)
  = (eq getTerm(processTerm(T1, M)) = getTerm(processTerm(T2, M)) [none] .) .
eq solveBubblesAux('__['eq_=_.[T1, T2], T3], M)
  = (eq getTerm(processTerm(T1, M)) = getTerm(processTerm(T2, M)) [none] .
     solveBubblesAux(T3, M)) .
ceq solveBubblesAux('__[F[TL], T2], M)
  = solveBubblesAux(T2, M)
  if F =/= 'eq_=_. .
ceq solveBubblesAux(F[TL], M)
  = none
  if F =/= '__ /\ F =/= 'eq_=_. .

op processTerm : Term Module ~> ResultPair .
eq processTerm('bubble[T], M)
  = metaParse(M, downTerm(T, nil), anyType) .

op addSort : FModule Sort -> FModule .
eq addSort(fmod QI is IL sorts SS . SsS ODS MbS EqS endfm, S)
```

```
   = fmod QI is IL sorts SS ; S . SsS ODS MbS EqS endfm .

  op addOpDecl : FModule OpDecl -> FModule .
  eq addOpDecl(fmod QI is IL sorts SS . SsS ODS MbS EqS endfm, OD)
   = fmod QI is IL sorts SS . SsS (OD ODS) MbS EqS endfm .

  op addImport : FModule Import -> FModule .
  eq addImport(fmod QI is IL sorts SS . SsS ODS MbS EqS endfm, I)
   = fmod QI is IL I sorts SS . SsS ODS MbS EqS endfm .

  op addEquations : FModule EquationSet -> FModule .
  eq addEquations(fmod QI is IL sorts SS . SsS ODS MbS EqS endfm, EqS')
   = fmod QI is IL sorts SS . SsS ODS MbS (EqS EqS') endfm .
endfm
```

We have then the following reductions:

```
Maude> red in MINI-MAUDE :
           extractSignature(
             getTerm(metaParse(upModule('MINI-MAUDE-SYNTAX, false),
              'fmod 'NAT3 'is
                 'op 's_ ': 'Nat3 '-> 'Nat3 '.
                 'sort 'Nat3 '.
                 'op '0 ': '-> 'Nat3 '.
                 'eq 's 's 's '0 '= '0 '.
              'endfm,
              'PreModule))) .
result FModule: fmod 'NAT3 is
  nil
  sorts 'Nat3 .
  none
  op '0 : nil -> 'Nat3 [none] .
  op 's_ : 'Nat3 -> 'Nat3 [none] .
  none
  none
endfm

Maude> red in MINI-MAUDE :
           processModule(
             getTerm(metaParse(upModule('MINI-MAUDE-SYNTAX, false),
              'fmod 'NAT3 'is
                 'sort 'Nat3 '.
                 'op 's_ ': 'Nat3 '-> 'Nat3 '.
                 'op '0 ': '-> 'Nat3 '.
                 'eq 's 's 's '0 '= '0 '.
              'endfm,
              'PreModule))) .
result FModule: fmod 'NAT3 is
  nil
  sorts 'Nat3 .
  none
  op '0 : nil -> 'Nat3 [none] .
  op 's_ : 'Nat3 -> 'Nat3 [none] .
  none
  eq 's_['s_['s_['0.Nat3]]] = '0.Nat3 [none] .
```

```
endfm

Maude> red in MINI-MAUDE :
           processModule(
             getTerm(metaParse(upModule('MINI-MAUDE-SYNTAX, false),
              'fmod 'NAT3 'is
                 'sort 'Nat3 '.
                 'op 's_ ': 'Nat3 '-> 'Nat3 '.
                 'op '0 ': '-> 'Nat3 '.
                 'eq 's 's 's 'N:Nat3 '= 'N:Nat3 '.
              'endfm,
              'PreModule))) .
result FModule: fmod 'NAT3 is
  nil
  sorts 'Nat3 .
  none
  op '0 : nil -> 'Nat3 [none] .
  op 's_ : 'Nat3 -> 'Nat3 [none] .
  none
  eq 's_['s_['s_['N:Nat3]]] = 'N:Nat3 [none] .
endfm
```

18.4 The LOOP-MODE module (deprecated)

Using object-oriented concepts, we specify in Maude a general input/output facility provided by the LOOP-MODE module shown below, which extends the module QID-LIST (see Section 7.10), into a generic read-eval-print loop.

```
mod LOOP-MODE is
  protecting QID-LIST .
  sorts State System .
  op [_,_,_] : QidList State QidList -> System [ctor special (...)] .
endm
```

The operator [_,_,_] can be seen as an object—that we call the *loop object*—with an input stream (the first argument), an output stream (the third argument), and a state (given by its second argument). This read-eval-print loop provided by LOOP-MODE is a simple mechanism that is no longer maintained. The support for communication with external objects (see Section 9) makes it possible to develop more general and flexible solutions for dealing with input/output.

Since with the loop object only one input stream is supported (the current terminal), the way to distinguish the input passed to the loop object from the input passed to the Maude system—either modules or commands—is by enclosing them in parentheses. When something enclosed in parentheses is written after the Maude prompt, it is converted into a list of quoted identifiers. This is done by the system by first breaking the input stream into a sequence of Maude identifiers (see Section 3.1) and then converting each of these identifiers into a quoted identifier by putting a quote in front of it, and appending the results into a list of quoted identifiers, which is then placed in the first slot of the loop object.[3] The output is handled in the reverse way, that is, the list of quoted identifiers placed in the third slot of the loop object is displayed on the terminal after applying the inverse process of "unquoting" each of

[3]The tokenize function tokenizes strings into lists of quoted identifiers (see Section 7.11).

the identifiers in the list.[4] However, the output stream is not cleared at the time when the output is printed; it is instead cleared when the next input is entered. We can think of the input and output events as *implicit rewrites* that transfer—in a slightly modified, quoted or unquoted form—the input and output data between two objects, namely the loop object and the "user" or "terminal" object.

Besides having input and output streams, terms of sort `System` give us the possibility of maintaining a state in their second component. This state has been declared in a completely generic way. In fact, the sort `State` in `LOOP-MODE` does not have any constructors. This gives complete flexibility for defining the terms we want to have for representing the state of the loop in each particular application. In this way, we can use this input/output facility not only for building user interfaces for applications written in Maude, but also for uses of Maude as a *metalanguage*, where the object language being implemented may be completely different from Maude. For each such tool or language the nature of the state of the system may be completely different. We can tailor the `State` sort to any such application by importing `LOOP-MODE` in a module in which we define the state structure and the rewrite rules for changing the state and interacting with the loop.

[4] The `printTokens` function takes a list of quoted identifiers and converts into a string with the usual printing conventions (see Section 7.11).

Chapter 19

Meta-interpreters

The `META-LEVEL` module is *purely functional*. This is because all its *descent* functions are deterministic, even though they may manipulate intrinsically nondeterministic entities such as rewrite theories. For example, the `metaSearch` descent function with a bound of, say, 3, is entirely deterministic, since given the meta-representations $\overline{\mathcal{R}}$ of the desired system module and $\overline{t}$ of the initial term plus the bound 3, the result yielded by `search` for $\mathcal{R}$, t and 3 at the object level, and therefore by `metaSearch` at the metalevel, is uniquely determined.

Although `META-LEVEL` is very powerful, its purely functional nature means that it has *no notion of state*. Therefore, reflective applications where *user interaction* in a state-changing manner is essential require using `META-LEVEL` in the context of additional features supporting such an interaction. Maude's *meta-interpreters* feature makes possible very flexible kinds of reflective interactions in which Maude interpreters are encapsulated as external objects and can reflectively interact with both other Maude interpreters and with various other external objects, including the user.

19.1 Maude meta-interpreters

Conceptually a meta-interpreter is an external object that is an independent Maude interpreter, complete with module and view databases, which sends and receives messages. The module `META-INTERPRETER` in the `meta-interpreter.maude` file contains command and reply messages that cover almost the entirety of the Maude interpreter. For example, it can be instructed to insert or show modules and views, or carry out computations in a named module. As response, the meta-interpreter replies with messages acknowledging operations carried out or containing results. Meta-interpreters can be created and destroyed as needed, and because a meta-interpreter is a complete Maude interpreter, it can host meta-interpreters itself and so on in a tower of reflection. Furthermore, the original `META-LEVEL` functional module can itself be used from inside a meta-interpreter after it is inserted.

Internally, the guts of the Maude interpreter implementation are encapsulated in a C++ class called `Interpreter` and the top-level interpreter that one interacts with on the command line is an instance of this class together with a small amount of glue code that enables it to communicate over the standard I/O streams. Meta-interpreters are also instances of this class, with a small amount of glue code that enables them to exchange messages with an enclosing object-oriented rewriting execution context. Currently, both the object-level interpreter and any existing meta-interpreters all run the same single threaded process, and control flow is managed through the object-oriented rewriting mechanism.

The metarepresentation of terms, modules, and views is shared with the `META-LEVEL` functional module in Chapter 17. The API to meta-interpreters defined in the `META-INTERPRETER` module includes several sorts and constructors, a built-in object identifier `interpreterManager`, and a large collection of command and response messages (see file `meta-interpreter.maude` for the complete details).

```
mod META-INTERPRETER is
  protecting META-LEVEL .
  including CONFIGURATION .

  sort RewriteCount .
  subsort Nat < RewriteCount .
  sorts InterpreterOption InterpreterOptionSet .
  subsort InterpreterOption < InterpreterOptionSet .

  op none : -> InterpreterOptionSet [ctor] .
  op interpreter : Nat -> Oid [ctor] .

  op createInterpreter : Oid Oid InterpreterOptionSet -> Msg [ctor msg ...] .
  op createdInterpreter : Oid Oid Oid -> Msg [ctor msg ...] .

  op insertModule : Oid Oid Module -> Msg [ctor msg ...] .
  op insertedModule : Oid Oid -> Msg [ctor msg ...] .
  op insertView : Oid Oid View -> Msg [ctor msg ...] .
  op insertedView : Oid Oid -> Msg [ctor msg ...] .
   ...
   ...
  op erewriteTerm : Oid Oid Bound Nat Qid Term -> Msg [ctor msg ...] .
  op erewroteTerm : Oid Oid RewriteCount Term Type -> Msg [ctor msg ...] .
   ...
   ...
  op quit : Oid Oid -> Msg [ctor msg ...] .
  op bye : Oid Oid -> Msg [ctor msg ...] .
  op interpreterManager : -> Oid [special (...)].
endm
```

All messages in this module follow the standard Maude message format, with the first two arguments being the object identifiers of the target and of the sender. The `interpreterManager` object identifier refers to a special external object that is responsible for creating new meta-interpreters in the current execution context. Such meta-interpreters have object identifiers of the form `interpreter(`n`)` for any natural number n.

19.2 A Russian dolls example

Let us illustrate the flexibility and generality of meta-interpreters with a short example. The example, which we call `RUSSIAN-DOLLS` after the Russian nesting dolls, performs a computation in a meta-interpreter that itself exists in a tower of meta-interpreters nested to a user-definable depth and requires only two equations and two rules.

```
mod RUSSIAN-DOLLS is
  extending META-INTERPRETER .

  op me : -> Oid .
```

```
  op User : -> Cid .
  op depth:_ : Nat -> Attribute .
  op computation:_ : Term -> Attribute .

  vars X Y Z : Oid .
  var AS : AttributeSet .
  var N : Nat .
  var T : Term .

  op newMetaState : Nat Term -> Term .
  eq newMetaState(0, T) = T .
  eq newMetaState(s N, T)
   = upTerm(
       <>
       < me : User | depth: N, computation: T >
       createInterpreter(interpreterManager, me, none)) .

  rl < X : User | AS >
     createdInterpreter(X, Y, Z)
  => < X : User | AS >
     insertModule(Z, X, upModule('RUSSIAN-DOLLS, true)) .
  rl < X : User | depth: N, computation: T, AS >
     insertedModule(X, Y)
  => < X : User | AS >
     erewriteTerm(Y, X, unbounded, 1, 'RUSSIAN-DOLLS, newMetaState(N, T)) .
endm
```

The visible state of the computation resides in a Maude object of identifier `me` and class `User`. The object holds two values in respective attributes: the depth of the meta-interpreter, which is recorded as a `Nat`, with `0` as the top level, and the computation to perform, which is recorded as a `Term`.

The operator `newMetaState` takes a depth and a metaterm to evaluate. If the depth is zero, then it simply returns the metaterm as the new metastate; otherwise, a new configuration is created, consisting of a portal (needed for rewriting with external objects to locate where messages exchanged with external objects leave and enter the configuration), the user-visible object holding the decremented depth and computation, and a message addressed to the `interpreterManager` external object, requesting the creation of a new meta-interpreter, and this configuration is lifted to the metalevel using the built-in `upTerm` operator imported from the functional metalevel.

The first rule of the `RUSSIAN-DOLLS` module handles the `createdInterpreter` message from `interpreterManager`, which carries the object identifier of the newly created meta-interpreter. It uses `upModule` to lift its own module, `RUSSIAN-DOLLS`, to the metalevel and sends a request to insert this meta-module into the new meta-interpreter. The second rule handles the `insertedModule` message from the new meta-interpreter. It calls the `newMetaState` operator to create a new metastate and then sends a request to the new meta-interpreter to perform an unbounded number of rewrites, with external object support and one rewrite per location per traversal in the metalevel copy of the `RUSSIAN-DOLLS` module that was just inserted.

We start the computation with an `erewrite` command on a configuration that consists of a portal, a user object, and a `createInterpreter` message:

```
Maude> erewrite
        <>
        < me : User | depth: 0, computation: ('_+_['s_^2['0.Zero], 's_^2['0.Zero]]) >
```

```
        createInterpreter(interpreterManager, me, none) .
result Configuration:
  <>
  < me : User | none >
  erewroteTerm(me, interpreter(0), 1, 's_^4['0.Zero], 'NzNat)
```

With depth 0, this results in the evaluation of the meta-representation of $2 + 2$ directly in a meta-interpreter, with no nesting. Passing a depth of 1 results in the evaluation instead being done in a nested meta-interpreter.

```
Maude> erewrite
        <>
        < me : User | depth: 1,
           computation: ('_+_['s_^2['0.Zero],'s_^2['0.Zero]]) >
        createInterpreter(interpreterManager, me, none) .
result Configuration:
  <>
  < me : User | none >
  erewroteTerm(me, interpreter(0), 5,
    '__['<>.Portal,
    '<_:_|_>['me.Oid,'User.Cid,'none.AttributeSet],
    'erewroteTerm['me.Oid,'interpreter['0.Zero],'s_['0.Zero],
    '_`[_`]['`'s_^4.Sort,''0.Zero.Constant],''NzNat.Sort]], 'Configuration)
```

Notice here that the top-level reply message `erewroteTerm(...)` contains a result that is a metaconfiguration, which contains the reply `'erewroteTerm[...]` metamessage from the inner meta-interpreter.

Additional examples of the use of standard streams and meta-interpreters can be found in the next section, where they are used to develop a simple execution environment.

19.3 An execution environment for Mini-Maude using IO and meta-interpreters

We illustrate in this section the use of standard streams and meta-interpreters to develop programming environments for our languages. Specifically, we present an environment for the MiniMaude language introduced in Section 18.3. The same techniques may be used to develop your language of choice. Once you have defined a grammar (`MINI-MAUDE-SYNTAX` in the case of MiniMaude) and a transformation from parse terms to Maude (as provided by the `processModule` operation in the `MINI-MAUDE` module) you can build an execution environment using the techniques shown in this section.

The MiniMaude language has been designed to be as simple as possible, but there were several features we wanted it to include:

- We can declare sorts and operations, and specify equations on the terms we can construct with them.
- We can include modules previously defined. We illustrate how we can store modules in the meta-interpreters database so that they can later be referred to or used.
- We can reduce terms to their normal forms using the equations in a module. The syntax defined for MiniMaude only allows the `reduce` command.

As pointed out in the previous section, the main difference between the metalevel and a meta-interpreter is that the metalevel is functional but the meta-interpreter is not, and we can interact with it through messages. Being non-functional, meta-interpreters provide functionality, for instance, to store modules and views in its database, and then operate on them. We will use that functionality in this section to illustrate its use and some of its possibilities. Indeed, meta-interpreters provide the same functionality as the metalevel, plus some additional features to insert elements in its database and retrieve them. In other words, we can say that it provides the desired functionality at both the object level and at the metalevel.

In our case, we will store a minimum of information in an object that will request inputs from the user using the standard input stream and will attempt to parse it in the MiniMaude grammar. To be able to parse inputs using the meta-interpreter, we will start by introducing the `MINI-MAUDE-SYNTAX` module in it. Once it is there, we can try to parse the inputs. When the standard stream receives a `getLine` message, it responds with the string typed by the user until a return key is pressed. To be able to parse multi-line inputs, we will need to request new lines until the input is completed. Of course, at any time we may get a parse error or an ambiguity, in which cases we need to report the given error.

Once the input is completed, we may have a module or a `reduce` command. If it is a module, we need to extract the signature first and then to solve the bubbles in its equations. The process must be carried out in two steps, since the signature may refer to submodules in the database of the meta-interpreter, but the equations with the bubbles processed are to be inserted in the top module. If the input corresponds to a `reduce` command, the term must be parsed, and then reduced by the meta-interpreter.

Although the process is quite systematic, different cases must be taken into account. To simplify the processing, we use an attribute `state` that keeps track of the different alternatives. Figure 19.1 shows a state diagram of the execution environment. This object also keeps the identifier of the meta-interpreter, the name of the last entered module, and the partial input introduced.

```
view Oid from TRIV to CONFIGURATION is
  sort Elt to Oid .
endv

view Module from TRIV to META-MODULE is
  sort Elt to Module .
endv

mod MINI-MAUDE-META-INTERPRETER is
  protecting MINI-MAUDE .
  including STD-STREAM .
  including LEXICAL .
  including MAYBE{Oid} * (op maybe to null) .
  including MAYBE{Qid} * (op maybe to null) .
  including MAYBE{Module} * (op maybe to null) .
  including META-INTERPRETER .

  vars O O' MI Y : Oid .
  var  Atts : AttributeSet .
  var  QIL : QidList .
  var  Str : String .
  vars T T' : Term .
  vars Ty Ty' : Type .
  var  N : Nat .
```

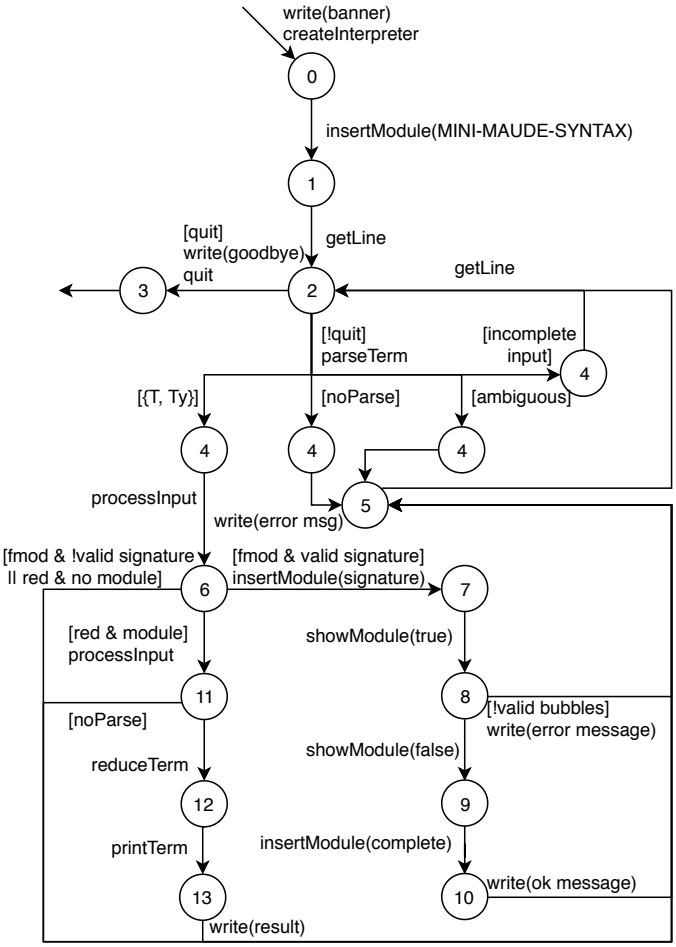


Figure 19.1: MiniMaude's statechart.

```
var RP? : [ResultPair] .
var M : Module .
var M? : Maybe{Module} .
vars QI? : Maybe{Qid} .
vars QI MN : Qid .
var EqS : EquationSet .
```

`MiniMaude` objects are represented using the following declarations.

```
sort MiniMaude .
subsort MiniMaude < Cid .
op MiniMaude : -> MiniMaude .
op mi:_ : Maybe{Oid} -> Attribute [prec 20 gather (&)] . ---- meta-interpreter
op mn:_ : Maybe{Qid} -> Attribute [prec 20 gather (&)] . ---- default module name
op in:_ : QidList -> Attribute [prec 20 gather (&)] .    ---- accumulated input
op st:_ : Nat -> Attribute [prec 20 gather (&)] .        ---- state
```

Several messages are used for intermediate steps.

```
op processInput : Oid Term -> Msg .
op pendingBubbles : Oid Term -> Msg .
op parsedEquations : Oid EquationSet -> Msg .
op processReduce : Oid [ResultPair] -> Msg .
```

The MiniMaude environment may be initiated using the `minimaude` constants, with it an interpreter is created and a banner is sent to the output stream.

```
op o : -> Oid .
op minimaude : -> Configuration .

eq minimaude
 = <>
   < o : MiniMaude | mi: null, mn: null, in: nil, st: 0 >
   write(stdout, o, "'\n\t MiniMaude Execution Environment\n")
   createInterpreter(interpreterManager, o, none) .
```

Once the message is written and the meta-interpreter created, the `MINI-MAUDE-SYNTAX` module is inserted in the meta-interpreter. The second argument of the `createdInterpreter` message is, as usual, the sender (`interpreterManager` in this case).

```
rl < O : MiniMaude | mi: null, st: 0, Atts >
   wrote(O, O')
   createdInterpreter(O, Y, MI)
=> < O : MiniMaude | mi: MI, st: 1, Atts >
   insertModule(MI, O, upModule('MINI-MAUDE-SYNTAX, true)) .
```

Once the module is inserted, a `getLine` message is sent to the `stdin` object.

```
rl < O : MiniMaude | mi: MI, st: 1, Atts >
   insertedModule(O, O')
=> < O : MiniMaude | mi: MI, st: 2, Atts >
   getLine(stdin, O, "minimaude> ") .
```

When the user introduces some inputs, the `stdin` object responds with a `gotLine` message with the string entered. The user is expected to write `quit` or `q` to leave the environment. If the input is one of these, a goodbye message is sent to the `stdout` object and the meta-interpreter is killed. Otherwise, an attempt is made to parse the input. Note that some previous inputs may be stored in the `in` attribute, so the message to the meta-interpreter to parse the input includes the entire `Qid` list.

```
rl < O : MiniMaude | mi: MI, in: QIL, st: 2, Atts >
   gotLine(O, O', Str)
=> if tokenize(Str) == 'quit or tokenize(Str) == 'q
   then < O : MiniMaude | mi: MI, in: nil, st: 3, Atts >
        write(stdout, o, "goodbye\n")
        quit(MI, O)
   else < O : MiniMaude | mi: MI, in: QIL tokenize(Str), st: 4, Atts >
        parseTerm(MI, O, 'MINI-MAUDE-SYNTAX, none, QIL tokenize(Str), 'Input)
   fi .
```

If a `quit` message was sent to the meta-interpreter, it would respond with a `bye` message. This is the final rule, which terminates the execution.

```
rl < O : MiniMaude | mi: MI, st: 3, Atts >
   wrote(O, O')
   bye(O, MI)
=> none .
```

If the parse succeeded, the meta-interpreter responds with a `parsedTermÂ` message that includes a term of sort `ResultPair`. This term in that pair is sent in a `processInput` message.

```
rl < O : MiniMaude | mi: MI, in: QIL, st: 4, Atts >
   parsedTerm(O, MI, {T, Ty})
=> < O : MiniMaude | mi: MI, in: nil, st: 6, Atts >
   processInput(O, T) .
```

If the parse failed, the `parsedTerm` message from the meta-interpreter includes a `noParse` term with the position at which the parsing failed. If the position is the end of the input it means that the input was incomplete, and in that case that partial input is added to the current input and additional text is requested from the user. If the error was in some other position, an error message is sent.

```
rl < O : MiniMaude | mi: MI, in: QIL, st: 4, Atts >
   parsedTerm(O, MI, noParse(N))
=> if N == size(QIL)
   then < O : MiniMaude | mi: MI, in: QIL, st: 2, Atts >
        getLine(stdin, O, "> ")
   else < O : MiniMaude | mi: MI, in: nil, st: 5, Atts >
        write(stdout, o, "Parse error\n")
   fi .
```

The response may also be informing about an ambiguity. Although more precise information might be given to the user, we have simplified the error message.

```
rl < O : MiniMaude | mi: MI, in: QIL, st: 4, Atts >
   parsedTerm(O, MI, ambiguity({T, Ty}, {T', Ty'}))
=> < O : MiniMaude | mi: MI, in: nil, st: 5, Atts >
   write(stdout, o, "Ambiguous input\n") .
```

The `processInput` message may correspond to a functional module or to a `reduce` command. In the first case, if the input contains a valid signature, such a module is inserted in the meta-interpreter. The input is kept in a `pendingBubbles` message for later processing.

```
rl < O : MiniMaude | mi: MI, mn: QI?, st: 6, Atts >
   processInput(O, 'fmod_is_endfm['token[T], T'])
=> if extractSignature('fmod_is_endfm['token[T], T']) :: Module
   then < O : MiniMaude | mi: MI, mn: downTerm(T, 'default-name), st: 7, Atts >
        insertModule(MI, O, extractSignature('fmod_is_endfm['token[T], T']))
        pendingBubbles(O, T')
   else < O : MiniMaude | mi: MI, mn: QI?, st: 5, Atts >
        write(stdout, o, "Parse error\n")
   fi .
```

Once the signature is inserted, the `MiniMaude` object requests the flattened module. If the module contains importations of previously entered modules, the entire module must be used for the processing of the bubbles in the top module.

```
rl < O : MiniMaude | mi: MI, mn: QI, st: 7, Atts >
   insertedModule(O, MI)
=> < O : MiniMaude | mi: MI, mn: QI, st: 8, Atts >
   showModule(MI, O, QI, true) .
```

The retrieved flattened module is then used to process the bubbles in the equations of the module. If the processing fails, an error message is shown to the user. If it succeeded, the top module is requested so that the equations can be added to it.

```
rl < O : MiniMaude | mi: MI, mn: QI, st: 8, Atts >
   showingModule(O, MI, M)
   pendingBubbles(O, T)
=> if solveBubblesAux(T, M) :: EquationSet
   then < O : MiniMaude | mi: MI, mn: QI, st: 9, Atts >
        parsedEquations(O, solveBubblesAux(T, M))
        showModule(MI, O, QI, false)
   else < O : MiniMaude | mi: MI, mn: QI, st: 5, Atts >
        write(stdout, o, "Parse error\n")
   fi .
```

The top module is inserted in the meta-interpreter once the processed equations are added to it.

```
rl < O : MiniMaude | mi: MI, mn: QI, st: 9, Atts >
   showingModule(O, MI, M)
   parsedEquations(O, EqS)
=> < O : MiniMaude | mi: MI, mn: QI, st: 10, Atts >
   insertModule(MI, O, addEquations(M, EqS)) .
```

When the insertion is completed with success, the user is informed.

```
rl < O : MiniMaude | mi: MI, st: 10, Atts >
   insertedModule(O, MI)
=> < O : MiniMaude | mi: MI, st: 5, Atts >
   write(stdout, O, "Module loaded successfully\n") .
```

As can be seen in Figure 19.1, state 5 is the one to which the object returns every time an operation concludes, either with a failure or success, after notifying the user, and then requesting further inputs.

```
rl < O : MiniMaude | mi: MI, st: 5, Atts >
   wrote(O, O')
=> < O : MiniMaude | mi: MI, st: 2, Atts >
   getLine(stdin, O, "minimaude> ") .
```

The input may also correspond to a `reduce` command. The following rule handles the case in which a command is introduced but there is no previous module inserted on which the command can be evaluated.

```
rl < O : MiniMaude | mn: null, st: 6, Atts >
   processInput(O, 'reduce_.[T])
=> < O : MiniMaude | mn: null, st: 5, Atts >
   write(stdout, o, "No module in the system\n") .
```

If there is a previous module, the term to be reduced must first be parsed.

```
rl < O : MiniMaude | mi: MI, mn: MN, st: 6, Atts >
   processInput(O, 'reduce_.['bubble[T]])
=> < O : MiniMaude | mi: MI, mn: MN, st: 11, Atts >
   parseTerm(MI, O, MN, none, downTerm(T, nil), anyType) .
```

The parsing of the term may result in success or failure. In the first case, a `reduceTerm` message is sent to the meta-interpreter. In the second case, an error message is given.

```
rl < O : MiniMaude | mi: MI, mn: MN, st: 11, Atts >
   parsedTerm(O, MI, {T, Ty})
=> < O : MiniMaude | mi: MI, mn: MN, st: 12, Atts >
   reduceTerm(MI, O, MN, T) .
```

```
  rl < O : MiniMaude | mi: MI, mn: MN, st: 11, Atts >
     parsedTerm(O, MI, noParse(N))
  => < O : MiniMaude | mi: MI, st: 5, Atts >
     write(stdout, o, "Parse error\n") .
```

Once the simplification command is completed, the meta-interpreter sends back a `reducedTerm` message with the result of the execution. Before showing the result to the user, the pretty-printing of the term must be requested to the meta-interpreter.

```
  rl < O : MiniMaude | mi: MI, mn: MN, st: 12, Atts >
     reducedTerm(O, MI, N, T, Ty)
  => < O : MiniMaude | mi: MI, mn: MN, st: 13, Atts >
     reducedTerm(O, MI, N, T, Ty)
     printTerm(MI, O, MN, none, T, mixfix flat format number rat) .

  rl < O : MiniMaude | mi: MI, st: 13, Atts >
     reducedTerm(O, MI, N, T, Ty)
     printedTerm(O, MI, QIL)
  => < O : MiniMaude | mi: MI, st: 5, Atts >
     write(stdout, o, "result " + string(Ty) + ": " + printTokens(QIL) + "\n") .
endm
```

We can run the environment with the `minimaude` configuration.

```
Maude> erew minimaude .
erewrite in MINI-MAUDE-META-INTERPRETER : minimaude .

        MiniMaude Execution Environment
minimaude>
```

First, let us introduce a simple example, the `NAT3` module already used in Section 18.3.

```
> fmod NAT3 is
>    sort Nat3 .
>    op s_ : Nat3 -> Nat3 .
>    op 0 : -> Nat3 .
>    eq s s s 0 = 0 .
> endfm
Module loaded successfully
```

We can execute a simple command on that module as follows.

```
minimaude> reduce s s s s 0 .
result Nat3: s 0
```

A more interesting module refers to the previous one, extending it with a plus operation.

```
minimaude> fmod NAT3+ is
>    including NAT3 .
>    op _+_ : Nat3 Nat3 -> Nat3 .
>    eq 0 + N:Nat3 = N:Nat3 .
>    eq s N:Nat3 + M:Nat3 = s (N:Nat3 + M:Nat3) .
> endfm
Module loaded successfully

minimaude> reduce s s 0 + s 0 .
result Nat3: 0
```

If wrong inputs are inserted, an error message is provided.

```
minimaude> reduce foo .
Parse error
```

We can finally exit the environment with a `q` command.

```
minimaude> q
goodbye
rewrites: 643 in 10ms cpu (128883ms real) (62885 rewrites/second)
result Portal: <>
```

Chapter 20

Debugging and Troubleshooting

20.1 Debugging approaches

There are several approaches to debugging and optimizing Maude programs: tracing, term coloring, using the debugger, and using the profiler.

20.1.1 Tracing

The tracing facilities allow us to follow the execution of our specifications, that is, the sequence of rewrites or equational simplification reductions that take place. Tracing is turned on with the command

```
set trace on .
```

A log of the trace can be captured using *script* or xterm logging. This can then be studied using a text editor. Since the trace is usually voluminous, there are a number of trace options to control just what is traced. We refer to Section 23.8 for a complete list of tracing commands and options.

One of the most useful options is selective tracing:

```
set trace select on .
trace select foo bar ([_,_]) .
```

This will cause only rewrites where the statement (equation, membership or rule) is labeled with a selected name or the redex is headed by operators with a selected name to be traced. In the above example, suppose `foo` and `bar` are rule labels, `[_,_]` is an operator name, and `foo` is also an operator name. Then, rewrites using the rules labeled by `foo` or `bar` will be reported, as will also rewrites with redex whose top-level operator is either `foo` or `[_,_]`. Note that these labels or operators need not be in existence at the time the `trace select` command is executed; thus it is possible to select statements and operators that will only be created at runtime via the metalevel.

A useful option for metaprogramming is

```
trace exclude FOO BAR .
```

This will exclude the named modules from being traced and thus allows one to selectively avoid tracing the chosen object and/or metalevel modules. This is particularly useful when using Full Maude to localize the tracing to the "object modules" being executed and *not* to the `FULL-MAUDE` module itself (see Chapter 21). After loading Full Maude, its specification is

excluded from the tracing, which allows us to trace Full Maude specifications as Core Maude specifications.

As we have mentioned, there are different commands that may help us in the control of the trace of the execution at hand. If the number of rewrites is small, we may use the whole trace to check the behavior of our specification. However, the number of rewrites is usually big, and considering the whole trace is completely impossible. The different options may help us, for example, to focus on a particular rule or set of rules, exclude certain modules from the trace, or not tracing the rewrites happening in the conditions.

Let us illustrate some of these commands to trace the bank accounts example presented in Section 8.1. To see the trace we just need to set the trace on. After it, the trace of any rewrite command will be given, according to the active options. By default, the application of every equation, membership axiom, and rule will be printed, showing the corresponding substitution, the current whole term, and the subterm on which the axiom is being applied before and after its application. To get a flavor of the information we get, let us rewrite the `bankConf` term with a bound of 1.

```
Maude> set trace on .
Maude> rew [1] bankConf .
rewrite [1] in BANK-ACCOUNT-TEST : bankConf .
********** equation
eq bankConf = (((((< A-003 : Account | bal : 1250 > from A-003 to
    A-002 transfer 300) debit(A-002, 400)) < A-002 : Account | bal :
    250 >) debit(A-001, 150)) debit(A-001, 200)) < A-001 : Account |
    bal : 300 > .
empty substitution
bankConf
--->
(((((< A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
    300) debit(A-002, 400)) < A-002 : Account | bal : 250 >) debit(
    A-001, 150)) debit(A-001, 200)) < A-001 : Account | bal : 300 >
********** trial #1
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
    : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true
    [label debit] .
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
********** solving condition fragment
N:Nat >= M:Nat = true
********** equation
(built-in equation for symbol _>=_)
300 >= 150
--->
true
********** success for condition fragment
N:Nat >= M:Nat = true
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
********** success #1
********** rule
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
    : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true
```

```
    [label debit] .
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
    Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
    Account | bal : 1250 > from A-003 to A-002 transfer 300
--->
(debit(A-001, 200) debit(A-002, 400) < A-002 : Account | bal : 250 >
    < A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
    300) < A-001 : Account | bal : (300 - 150) >
********** equation
(built-in equation for symbol _-_)
300 - 150
--->
150
rewrites: 4 in 1ms cpu (1ms real) (4000 rewrites/second)
result Configuration: debit(A-001, 200) debit(A-002, 400) < A-001 :
    Account | bal : 150 > < A-002 : Account | bal : 250 > < A-003 :
    Account | bal : 1250 > from A-003 to A-002 transfer 300
```

Notice that, even though the bound for the rewrite command is one, there are four rewrites. Recall that the bound only concerns rule application. In this trace we see how three equations—two of them built-in—are also applied. In addition to the statement used in each rewriting step, the trace shows the matching substitution and the whole term, before and after the application of the statement. Notice also the information concerning the evaluation of conditions. We can see that, although there is a match with the `debit` rule, this rule is not applied until the success of its condition has been checked.

Suppose we are mainly concerned with the application of rules. In this case we may think that there is too much "noise" due to the application of equations. We may request hiding the information about the application of equations with the command `set trace eq off`. Then the trace for rewriting the same `bankConf` term with the same bound of 1 is as follows:

```
Maude> set trace eq off .
Maude> rew [1] bankConf .
rewrite [1] in BANK-ACCOUNT-TEST : bankConf .
********** trial #1
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
    : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true
    [label debit] .
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
********** solving condition fragment
N:Nat >= M:Nat = true
********** success for condition fragment
N:Nat >= M:Nat = true
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
********** success #1
********** rule
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
    : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true
```

```
    [label debit] .
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
    Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
    Account | bal : 1250 > from A-003 to A-002 transfer 300
--->
(debit(A-001, 200) debit(A-002, 400) < A-002 : Account | bal : 250 >
    < A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
    300) < A-001 : Account | bal : (300 - 150) >
rewrites: 4 in 1ms cpu (0ms real) (4000 rewrites/second)
result Configuration: debit(A-001, 200) debit(A-002, 400) < A-001 :
    Account | bal : 150 > < A-002 : Account | bal : 250 > < A-003 :
    Account | bal : 1250 > from A-003 to A-002 transfer 300
```

The selection of the concrete operator or statement label to trace may also be a good alternative when looking for something specific. Suppose that we are suspicious of a particular rule, say `transfer`. We may get the applications of such a rule for the unbounded rewrite of the `bankConf` term by using the `trace select` command as follows.

```
Maude> set trace select on .
Maude> trace select transfer .
Maude> rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
********** trial #1
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
    N':Nat > fromA:Oid to B:Oid transfer M:Nat => < A:Oid : Account |
    bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
    N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
A:Oid --> A-003
N:Nat --> 1250
B:Oid --> A-002
N':Nat --> 250
M:Nat --> 300
********** solving condition fragment
N:Nat >= M:Nat = true
********** success for condition fragment
N:Nat >= M:Nat = true
A:Oid --> A-003
N:Nat --> 1250
B:Oid --> A-002
N':Nat --> 250
M:Nat --> 300
********** success #1
********** rule
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
    N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
    | bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
    N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
A:Oid --> A-003
N:Nat --> 1250
B:Oid --> A-002
N':Nat --> 250
M:Nat --> 300
```

```
debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >
    < A-002 : Account | bal : 250 > < A-003 : Account | bal : 1250 >
    from A-003 to A-002 transfer 300
--->
(debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >)
    < A-003 : Account | bal : (1250 - 300) > < A-002 : Account |
    bal : (300 + 250) >
rewrites: 13 in 1ms cpu (1ms real) (13000 rewrites/second)
result Configuration: debit(A-001, 200) < A-001 : Account |
    bal : 150 > < A-002 : Account | bal : 150 > < A-003 : Account |
    bal : 950 >
```

We may also hide some of the information being shown. For example, we may get the same trace without the substitutions being shown with the `set trace substitution off` command.

```
Maude> set trace substitution off .
Maude> rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
********** trial #1
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
    N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
    | bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
    N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
********** solving condition fragment
N:Nat >= M:Nat = true
********** success for condition fragment
N:Nat >= M:Nat = true
********** success #1
********** rule
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
    N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
    | bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
    N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >
    < A-002 : Account | bal : 250 > < A-003 : Account | bal : 1250 >
    from A-003 to A-002 transfer 300
--->
(debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >)
    < A-003 : Account | bal : (1250 - 300) > < A-002 : Account |
    bal : (300 + 250) >
rewrites: 13 in 0ms cpu (0ms real) (~ rewrites/second)
result Configuration: debit(A-001, 200) < A-001 : Account |
    bal : 150 > < A-002 : Account | bal : 150 > < A-003 : Account |
    bal : 950 >
```

Let us consider now a different example, namely, the `PATH` module presented in Sections 3.5 and 4.3. We use it here to illustrate the trace given for membership axioms and for conditional axioms with multiple fragments. We recall first the conditional membership axiom defining multi-edge paths and the conditional equation defining the associativity of path concatenation.

```
var  E : Edge .
vars P Q R S : Path .
cmb E ; P : Path if target(E) = source(P) .
ceq (P ; Q) ; R = P ; (Q ; R)
  if target(P) = source(Q) /\ target(Q) = source(R) .
```

Now we request the trace for the reduction of the term `length((b ; c) ; d)`. The information shown is particularly illustrative for understanding the way in which the membership axioms are used and the way conditions are evaluated. Note that the equation expressing the associativity of path concatenation has two fragments, one of which is evaluated after the other. In case the condition of a matching equation fails another equation is attempted; furthermore, equations with matching conditions have unbounded variables initially.

Since the full trace is more than six pages long, we use the `set trace condition off` command, so that the evaluation of the conditions is omitted.

```
Maude> set trace on .
Maude> set trace condition off .
Maude> red length((b ; c) ; d) .
reduce in PATH : length((b ; c) ; d) .
********** trial #1
cmb E ; P : Path if target(E) = source(P) .
E --> b
P --> c
********** solving condition fragment
target(E) = source(P)
********** success for condition fragment
target(E) = source(P)
E --> b
P --> c
********** success #1
********** membership axiom
cmb E ; P : Path if target(E) = source(P) .
E --> b
P --> c
[Path]: b ; c becomes Path
********** trial #2
ceq (P ; Q) ; R = P ; (Q ; R)
  if target(P) = source(Q) /\ target(Q) = source(R) .
P --> b
Q --> c
R --> d
********** solving condition fragment
target(P) = source(Q)
********** success for condition fragment
target(P) = source(Q)
P --> b
Q --> c
R --> d
********** solving condition fragment
target(Q) = source(R)
********** success for condition fragment
target(Q) = source(R)
P --> b
Q --> c
R --> d
********** success #2
********** equation
ceq (P ; Q) ; R = P ; (Q ; R)
  if target(P) = source(Q) /\ target(Q) = source(R) .
P --> b
```

```
Q --> c
R --> d
(b ; c) ; d
--->
b ; (c ; d)
********** trial #3
cmb E ; P : Path if target(E) = source(P) .
E --> c
P --> d
********** solving condition fragment
target(E) = source(P)
********** success for condition fragment
target(E) = source(P)
E --> c
P --> d
********** success #3
********** membership axiom
cmb E ; P : Path if target(E) = source(P) .
E --> c
P --> d
[Path]: c ; d becomes Path
********** trial #4
cmb E ; P : Path if target(E) = source(P) .
E --> b
P --> c ; d
********** solving condition fragment
target(E) = source(P)
********** success for condition fragment
target(E) = source(P)
E --> b
P --> c ; d
********** success #4
********** membership axiom
cmb E ; P : Path if target(E) = source(P) .
E --> b
P --> c ; d
[Path]: b ; (c ; d) becomes Path
********** trial #5
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> b
P --> c ; d
********** solving condition fragment
E ; P : Path
********** success for condition fragment
E ; P : Path
E --> b
P --> c ; d
********** success #5
********** equation
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> b
P --> c ; d
length(b ; (c ; d))
--->
```

```
1 + length(c ; d)
********** trial #6
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> c
P --> d
********** solving condition fragment
E ; P : Path
********** success for condition fragment
E ; P : Path
E --> c
P --> d
********** success #6
********** equation
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> c
P --> d
length(c ; d)
--->
1 + length(d)
********** equation
eq length(E) = 1 .
E --> d
length(d)
--->
1
********** equation
(built-in equation for symbol _+_)
1 + 1
--->
2
********** equation
(built-in equation for symbol _+_)
1 + 2
--->
3
rewrites: 20 in 2ms cpu (1ms real) (10000 rewrites/second)
result NzNat: 3
```

But the trace is too long to observe what we were interested in. Suppose we just wanted to check a possible mistake in the specification of the `length` function. We may select it for filtering the equations defining it.

```
Maude> set trace select on .
Maude> trace select length .
Maude> red length((b ; c) ; d) .
reduce in PATH : length((b ; c) ; d) .
********** trial #1
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> b
P --> c ; d
********** solving condition fragment
E ; P : Path
********** success for condition fragment
E ; P : Path
E --> b
```

```
P --> c ; d
********** success #1
********** equation
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> b
P --> c ; d
length(b ; (c ; d))
--->
1 + length(c ; d)
********** trial #2
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> c
P --> d
********** solving condition fragment
E ; P : Path
********** success for condition fragment
E ; P : Path
E --> c
P --> d
********** success #2
********** equation
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> c
P --> d
length(c ; d)
--->
1 + length(d)
********** equation
eq length(E) = 1 .
E --> d
length(d)
--->
1
rewrites: 20 in 0ms cpu (1ms real) (~ rewrites/second)
result NzNat: 3
```

20.1.2 Term coloring

A common failure mode of Maude programs is when a term does not fully reduce. This is a lack of sufficient completeness. For linear unconditional order-sorted specifications, sufficient completeness can be checked with the SCC tool [83]. However, for general Maude specifications proving sufficient completeness may require inductive theorem proving. If a term does not fully reduce, that is, if nonconstructor symbols remain in the term's canonical form, it can be difficult to determine just where the problem began, since when a subterm fails to reduce, the enclosing term often fails to reduce, and so on, leading to a large unreduced term. If the specification makes consistent use of the `ctor` attribute, problem subterms can be pinpointed by switching on term coloring with the command

```
set print color on .
```

Symbols within terms that are being executed (i.e., in a trace or in the final result of a `reduce` command) are colored as follows:

reduced, ctor	not colored
reduced, non-ctor, strangeness below	blue
reduced, non-ctor, no strangeness below	red
unreduced, no reduced above	green
unreduced, reduced directly above	magenta
unreduced, reduced not directly above	cyan

If an operator is colored, this means that the term contains nonconstructors, that is, that there is "strangeness" in the term. The different colors indicate the source of the strangeness. The idea is that red and magenta indicate the initial locus of a bug, while blue and cyan indicate secondary damage. Green denotes reduction pending and cannot appear in the final result. An example is the following module, in which there is a missing case in each of the definitions of the `_<_` and `min` operators (`0 < 0` and `min(N N)`, respectively).

```
fmod NAT-MSET-MIN is
  protecting BOOL .
  sorts Nat NatMSet .
  subsort Nat < NatMSet .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _ _ : NatMSet NatMSet -> NatMSet [assoc comm ctor] .
  op _<_ : Nat Nat -> Bool .
  op min : NatMSet -> Nat .

  vars N M : Nat .
  var  S : NatMSet .
  eq 0 < s(N) = true .
  eq s(N) < 0 = false .
  eq s(N) < s(M) = N < M .
  eq min(N N S) = min(N S) .
  ceq min(N M S) = min(N S) if N < M .
  ceq min(N M) = N if N < M .
  eq min(N) = N .
endfm
```

With color printing turned on, reducing `min(s(s(0)) s(s(0)))` returns the term with the `min` operator colored red, indicating a nonconstructor that can't be reduced. Reducing `min(s(s(0)) min(s(0) s(0)))` returns the term with the inner occurrence of the `min` operator colored red as above, and the outer occurrence colored blue, indicating that the problem probably lies in a subterm.

To avoid confusion, any colors that may have been specified using the `format` attribute (see Section 4.4.5) are ignored in this mode.

20.1.3 The debugger

There are three ways to get into the Maude debugger:

- a control-C interrupt during rewriting,
- prefixing a command with the keyword `debug`, and
- hitting a break point.

Break points are set with the command

```
break select foo bar ([_,_]) .
```

where the names refer to operators or statement (equation, membership, rule or strategy definition) labels in a way that is completely analogous to the `trace select` command described in Section 20.1.1. Break points are enabled with the command

```
set break on .
```

On entering the debugger, the prompt changes to `Debug(`n`)>` where n is the debug level, that is, the number of times the debugger has been re-entered (it is fully re-entrant). All top-level commands can be executed from the debugger, along with four commands that are special to the debugger:

`where .` Prints out the stack of pending rewrites, explaining how each one arose.

`step .` Executes the next rewrite with tracing turned on.

`resume .` Exits the debugger and continues with the current rewriting task.

`abort .` Exits the debugger and abandons the current rewriting task.

We illustrate these commands using the bank accounts example presented in Section 8.1 (assuming it is in the file `bank-account-test.maude`).

We first use the `debug` command to activate the debugger from the begining of a rewrite. Note the use of the `where`, `step`, and `resume` commands.

```
Maude> load bank-account-test.maude
Maude> debug rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
Debug(1)> where .
Current term is:
bankConf
which arose while executing a top level command.
Debug(1)> step .
********** equation
eq bankConf = (((((< A-003 : Account | bal : 1250 > from A-003 to
    A-002 transfer 300) debit(A-002, 400)) < A-002 : Account | bal :
    250 >) debit(A-001, 150)) debit(A-001, 200)) < A-001 : Account |
    bal : 300 > .
empty substitution
bankConf
--->
(((((< A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
    300) debit(A-002, 400)) < A-002 : Account | bal : 250 >) debit(
    A-001, 150)) debit(A-001, 200)) < A-001 : Account | bal : 300 >
Debug(1)> where .
Current term is:
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
    Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
    Account | bal : 1250 > from A-003 to A-002 transfer 300
which arose while executing a top level command.
Debug(1)> resume .
rewrites: 13 in 2ms cpu (85160ms real) (6500 rewrites/second)
result Configuration: debit(A-001, 200) < A-001 : Account | bal :
    150 > < A-002 : Account | bal : 150 > < A-003 : Account | bal :
    950 >
```

As said above, we can also enter into the debugger by reaching a break point or typing control-c. In the following example we set a break point on the `debit` rule, take a step, and then abort the rewrite process.

```
Maude> set break on .
Maude> break select debit .
Maude> rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
break on labeled rule:
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
    : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true [
    label debit] .
Debug(1)> where .
Current term is:
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
    Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
    Account | bal : 1250 > from A-003 to A-002 transfer 300
which arose while executing a top level command.
Debug(1)> step .
********** trial #1
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => <
    A:Oid : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat =
    true [label debit] .
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
********** solving condition fragment
N:Nat >= M:Nat = true
Debug(1)> where .
Current term is:
300 >= 150
which arose while checking a condition during the evaluation of:
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
  Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
  Account | bal : 1250 > from A-003 to A-002 transfer 300
which arose while executing a top level command.
Debug(1)> abort .
Maude>
```

Our last example illustrates the re-entering nature of the debugger. As said above, any command can be used during the debugging process, allowing, for example, starting an execution while debugging another one. We execute a `debug rew` command, entering the debugger, where we set a break point on the `transfer` rule. Notice the `Debug(2)>` prompt. Notice also how after getting out of the inner debugger the break point is still active.

```
Maude> debug rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
Debug(1)> step .
********** equation
eq bankConf = (((((< A-003 : Account | bal : 1250 > from A-003 to
    A-002 transfer 300) debit(A-002, 400)) < A-002 : Account | bal
    : 250 >) debit(A-001, 150)) debit(A-001, 200)) < A-001 : Account
    | bal : 300 > .
empty substitution
bankConf
```

```
--->
(((((< A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
    300) debit(A-002, 400)) < A-002 : Account | bal : 250 >) debit(
    A-001, 150)) debit(A-001, 200)) < A-001 : Account | bal : 300 >
Debug(1)> set break on .
Debug(1)> break select transfer .
Debug(1)> rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
break on labeled rule:
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
    N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
    | bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
    N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
Debug(2)> where .
Current term is:
debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >
    < A-002 : Account | bal : 250 > < A-003 : Account | bal : 1250 >
    from A-003 to A-002 transfer 300
which arose while executing a top level command.
Debug(2)> resume .
break on labeled rule:
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
  N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account |
  bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat + N':Nat)
  > if N:Nat >= M:Nat = true [label transfer] .
Debug(2)> abort .
Debug(1)> resume .
break on labeled rule:
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
    N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
    | bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
    N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
Debug(1)> set break off .
Debug(1)> resume .
rewrites: 13 in 4ms cpu (63920ms real) (2600 rewrites/second)
result Configuration: debit(A-001, 200) < A-001 : Account | bal :
    150 > < A-002 : Account | bal : 150 > < A-003 : Account | bal :
    950 >
```

20.1.4 Status report

We can find out what Maude is doing during execution without sending it a control-C (which will close files and mess up socket communication among other things since it has to exit blocked system calls to respond to the user).

Depending on whether Maude is running the status report may be activated either using `ctrl-T` or `kill -INFO <pid>` (Mac)[1] and `kill -USR1 <pid>` (Linux).[2] When activated, Maude will print out the current time, rewrite counts and a summary of what it is doing on `stderr`.

[1] On Mac, the status report feature uses the BSD SIGINFO extension, which is activated by `control-T` from a terminal where Maude is the foreground process. It can also be activated by `kill -INFO <pid>`, however this does not give the load and timing information as `control-T` does.

[2] On Linux, there is no equivalent capability so the Maude implementation follows the `dd` convention and `kill -USR1 <pid>` can be used instead.

20.1.5 The profiler

Tuning up of specifications is something that may be useful in many practical situations. We illustrate the use of the profiling facilities available in Maude to understand better the execution of our specifications, helping us in this way to make them more efficient. We will discuss the use of the profiler on two examples, namely, the specification of the Fibonacci function already discussed in Section 4.4.8 and the specification of sorted lists presented in Section 6.3.6.

First of all, it must be clear that there is no magic recipe on how to optimize our specifications. On the contrary, although there are some guidelines that we may try to follow when possible, it is not always the case that they work, or that they are applicable. For example, conditional rules are generally expensive from a computational point of view, as are membership axioms, but in some cases we may be interested in using proving tools for which using them could be a better alternative. Similarly, in Section 4.4.8 we saw that using the `memo` attribute was a big win in the case of the Fibonacci function, but it is not always applicable; for some specifications, the consumption of memory can become so big that we may be getting a slower specification. There is always a tradeoff between the speedup obtained using memoization and the amount of memory and the cost of handling it. We illustrate all these and other concerns in this section.

Profiling is switched on by the command `set profile on`. When profiling is switched on, a count of the number of executions of each statement (equation, membership, rule, and strategy definition) is kept. For unconditional statements, the profile information is just the number of rewrites using that statement. For conditional ones there is also the number of matches, since not every match leads to a rewrite, due to condition failure. Moreover, when searching there can be multiple rewrites for each match, since the condition may be solved in multiple ways. There is a table that for each condition fragment gives:

1. the number of times the fragment was initially tested,
2. the number of times the fragment was tested due to backtracking,
3. the number of times the fragment succeeded, and
4. the number of times the fragment failed.

Normally, (1) + (2) = (3) + (4).

Special rewrites such as built-in rewrites and memoized rewrites are also tracked, but these are associated with symbols rather than with statements. For conciseness, symbols with no special rewrites, and statements that are not matched are omitted. There are some limitations: metalevel rewrites are not displayed, due to the ephemeral nature of metamodules. In addition, condition fragments associated with a match or search command are not tracked (though any rewrites initiated by such a fragment are). If you turn profiling on or off in the debugger you may get inconsistent results.

The profile information is associated with each module and is usually cleared at the start of any command that can do rewrites, except `continue`. This behavior can be changed with the `set clear profile on / off` command.

Let us first consider the Fibonacci function described in Section 4.4.8.

```
fmod FIBONACCI is
  protecting NAT .
  op fibo : Nat -> Nat .

  var N : Nat .
  eq fibo(0) = 0 .
```

```
  eq fibo(1) = 1 .
  eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

Notice in the following reductions that the times given when the profile is active are slightly higher.

```
Maude> red fibo(30) .
reduce in FIBONACCI : fibo(30) .
rewrites: 4038805 in 3920ms cpu (3960ms real) (1030201 rews/sec)
result NzNat: 832040

Maude> set profile on .

Maude> red fibo(30) .
reduce in FIBONACCI : fibo(30) .
rewrites: 4038805 in 4170ms cpu (4194ms real) (968453 rews/sec)
result NzNat: 832040
```

After doing the reduction with the profiler activated, we can request the collected information by means of the command `show profile`. In this example, since the module has no memberships or rules and there are no conditional axioms, the profiler gives the number of times each of the equations has been applied and also the number of times built-in functions are called.

```
Maude> show profile .

op _+_ : [Nat] [Nat] -> [Nat] .
built-in eq rewrites: 1346268 (33.3333%)

eq fibo(0) = 0 .
rewrites: 514229 (12.7322%)

eq fibo(1) = 1 .
rewrites: 832040 (20.6011%)

eq fibo(s_^2(N)) = fibo(N) + fibo(s N) .
rewrites: 1346268 (33.3333%)
```

In this very simple example we observe that only the three equations in the `FIBONACCI` module plus the predefined addition operation on natural numbers have been used. We can also observe how the equations are applied a number of times relatively similar, with percentages 12, 20, and 33, respectively. More interesting is the number of times each of them is applied, which goes to 1346268 for the third equation. Taking into account that we reduced `fibo(30)`, it means that the calculations have been repeated many times. As we saw in Section 4.4.8, this is a good place to use the `memo` attribute: calculations on small arguments are repeated many times and a small amount of memory is needed for storing the result of such calculations.

After adding the `memo` attribute to the `fibo` operator, we get the following results from the profiler:

```
Maude> set profile on .

Maude> red fibo(30) .
reduce in FIBONACCI : fibo(30) .
rewrites: 88 in 1ms cpu (1ms real) (88000 rews/sec)
```

```
result NzNat: 832040

Maude> show profile .
op _+_ : [Nat] [Nat] -> [Nat] .
built-in eq rewrites: 29 (32.9545%)

op fibo : [Nat] -> [Nat] .
memo rewrites: 28 (31.8182%)

eq fibo(0) = 0 .
rewrites: 1 (1.13636%)

eq fibo(1) = 1 .
rewrites: 1 (1.13636%)

eq fibo(s_^2(N)) = fibo(N) + fibo(s N) .
rewrites: 29 (32.9545%)
```

As we already saw in Section 4.4.8, the number of rewrites and the time consumed in the computation have decreased dramatically. We may observe that now each of the values in the Fibonacci sequence is calculated only once.

Let us consider now another example, namely, the parameterized module `SORTED-LIST` presented in Section 6.3.6, which defines a sort `SortedList{X}` of sorted lists as a subsort of the sort `List{TOSET}{X}` of lists. In this case we deal with a parameterized module which imports several other modules, and which has membership axioms and equations, some of which are conditional.

First of all, notice that by default the profiler provides information on a particular computation. In this example, it is not the same sorting a list in reverse order as an already sorted list, and is not the same using insertion sort, mergesort, or quicksort for sorting. To have a better insight about our specification, and thus gaining chances of improving it, we should consider several reductions, dealing with different cases, the different sorting algorithms in our case.

To be able to run examples on big lists, with numbers initially sorted in different ways, let us consider the following module `NAT-LIST-GENERATOR`, which imports the module `SORTED-LIST {NatAsToset}` defining sorted lists of natural numbers, and specifies functions `nats-upto`, that builds lists from zero to the specified value, and `random-nats`, which generates a list of the specified number of random numbers.

```
fmod NAT-LIST-GENERATOR is
  protecting SORTED-LIST{NatAsToset} .
  protecting RANDOM .

  vars N M : Nat .

  op nats-upto : Nat -> NeSortedList{NatAsToset} .
  eq nats-upto(s N) = nats-upto(N) ++ s N : [] .
  eq nats-upto(0) = 0 : [] .

  op random-nats : Nat -> List{TOSET}{NatAsToset} .
  op random-nats : Nat Nat -> List{TOSET}{NatAsToset} .
  eq random-nats(N) = random-nats(0, N) .
  ceq random-nats(N, M)
    = random(N) : random-nats(s N, M)
    if N <= M .
```

```
  eq random-nats(N, M) = [] [owise] .
endfm
```

We execute each one of the `insertion-sort`, `mergesort`, and `quicksort` algorithms on three lists, namely, a sorted list, a list in reverse order, and a random one, each of them with 1000 elements.

```
Maude> red insertion-sort(nats-upto(1000)) .
rewrites: 2012009 in 1032ms cpu (1079ms real) (1948029 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

Maude> red insertion-sort(reverse(nats-upto(1000))) .
rewrites: 5519515 in 3634ms cpu (3694ms real) (1518667 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

Maude> red insertion-sort(random-nats(1000)) .
rewrites: 1535372 in 1286ms cpu (1397ms real) (1193166 rews/sec)
result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...

Maude> red mergesort(nats-upto(1000)) .
rewrites: 2082358 in 1079ms cpu (1134ms real) (1928402 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

Maude> red mergesort(reverse(nats-upto(1000))) .
rewrites: 2578581 in 1210ms cpu (1221ms real) (2129622 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

Maude> red mergesort(random-nats(1000)) .
rewrites: 88005 in 71ms cpu (77ms real) (1222478 rews/sec)
result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...

Maude> red quicksort(nats-upto(1000)) .
rewrites: 6519514 in 5065ms cpu (5344ms real) (1287111 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

Maude> red quicksort(reverse(nats-upto(1000))) .
rewrites: 6528518 in 4096ms cpu (4130ms real) (1593729 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

Maude> red quicksort(random-nats(1000)) .
rewrites: 97858 in 75ms cpu (84ms real) (1287791 rews/sec)
result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...
```

Instead of considering the profiling information separately, we use the `set clear profile off` command, so that the profiling information gets accumulated.

```
Maude> set clear profile off .

Maude> set profile on .

Maude> red insertion-sort(nats-upto(1000)) .
rewrites: 2012009 in 1169ms cpu (1351ms real) (1719927 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

Maude> red insertion-sort(reverse(nats-upto(1000))) .
...
```

As mentioned above, with the profiler active, the times taken by the reductions are slightly higher. The number of rewrites and cpu time for each of the cases is presented in the tables displayed in Figure 20.1 (page 483), where we also include these values for all the different executions discussed in the rest of the section.[3] The information shown by the profiler with the `show profile` command is three pages long; we just comment the most interesting pieces.

1. Predefined operators `_+_`, `_quo_`, `_<=_`, `_>_`, and `random` are used an important number of times, in particular `_<=_` and `_>_`, which are applied, respectively, 7599823 (28.2%) and 1770593 times (6.6%). Notice that `_<=_` is only used in the conditions of one of the membership axioms and in some of the conditions of the equations for `insert-list`, `merge`, `leq-elems`, `gr-elems`, and `random-nats`; the operator `_>_` is used in the conditions of the equations for `insert-list`, `mergesort`, `merge`, `leq-elems`, and `gr-elems`.

```
op _+_ : [Nat] [Nat] -> [Nat] .
built-in eq rewrites: 29961 (0.111124%)

op _quo_ : [Nat] [Nat] -> [Nat] .
built-in eq rewrites: 3000 (0.0111269%)

op _<=_ : [Nat] [Nat] -> [Bool] .
built-in eq rewrites: 7599823 (28.1874%)

op _>_ : [Nat] [Nat] -> [Bool] .
built-in eq rewrites: 1770593 (6.56706%)

op random : [Nat] -> [Nat] .
built-in eq rewrites: 3003 (0.011138%)
```

2. From the numbers for the membership axioms we may conclude that they are applied a considerable number of times, in particular the conditional one—4790561 rewrites (17.8%)—. Note that for conditional membership axioms, as for all conditional axioms, the system gives information on the number of matches, that is, the number of times that the conditions are reduced. It also provides the number of times each one of the fragments of the condition is reduced. In the case of the membership axioms in this specification, there is only one fragment. The part of the output corresponding to the membership axioms is the following:

```
mb [] : SortedList{NatAsToset} .
rewrites: 22040 (0.0817455%)

mb N : [] : NeSortedList{NatAsToset} .
rewrites: 17505 (0.0649254%)

cmb N : NEOL:NeSortedList{NatAsToset} : NeSortedList{NatAsToset}
  if N <= head(NEOL:NeSortedList{NatAsToset}) = true .
lhs matches: 4795972  rewrites: 4790561 (17.768%)
Fragment  Initial tries  Resolve tries  Successes   Failures
1         4795972        0              4790561     5411
```

We see that the condition has been checked 4795972 times, out of which only 5411 failed.

[3] All these figures have been obtained running Maude during a hot summer night on a Linux platform with an Intel Pentium M760 2GHz processor and 1GB of memory.

3. From the equations specifying the `insertion-sort` algorithm, the ones used more times are the two conditional ones for the `insert-list` function. From the information in the profile we see that these conditional equations have been attempted almost the same number of times, 756888 and 754895. We also see that both have been applied almost the same number of times, because the one that was attempted first almost always failed the evaluation of its condition, and then the second equation was applied.

```
eq insertion-sort([]) = [] .
rewrites: 3 (1.11269e-05%)

eq insertion-sort(N : L:List{TOSET}{NatAsToset}) = insert-list(
  insertion-sort(L:List{TOSET}{NatAsToset}), N) .
rewrites: 3003 (0.011138%)

eq insert-list([], M) = M : [] .
rewrites: 1010 (0.00374605%)

ceq insert-list(N : OL:SortedList{NatAsToset}, M) = M : N :
  OL:SortedList{NatAsToset} if M <= N = true .
lhs matches: 756888   rewrites: 1993 (0.00739196%)
Fragment  Initial tries  Resolve tries  Successes   Failures
1         756888         0              1993        754895

ceq insert-list(N : OL:SortedList{NatAsToset}, M) = N :
  insert-list(OL:SortedList{NatAsToset}, M) if M > N = true .
lhs matches: 754895   rewrites: 754895 (2.79988%)
Fragment  Initial tries  Resolve tries  Successes   Failures
1         754895         0              754895      0
```

4. The information on the equations for `mergesort` presents a similar pattern, the main difference being that there is only one conditional equation, since `merge` is declared as commutative. In this case the number of rewrites of all the equations is relatively small, much smaller than the total of rewrites in the computation. This makes us think that the weight of the computation of the memberships and generation of the lists to sort is much higher that the sorting itself.

```
eq mergesort(N : []) = N : [] .
rewrites: 3003 (0.011138%)

ceq mergesort(L:List{TOSET}{NatAsToset}) = merge(mergesort(take
  length(L:List{TOSET}{NatAsToset}) quo 2 from L:List{TOSET}{
  NatAsToset}),mergesort(throw length(L:List{TOSET}{
  NatAsToset}) quo 2 from L:List{TOSET}{NatAsToset})) if
  length(L:List{TOSET}{NatAsToset}) > 1 = true .
lhs matches: 3000     rewrites: 3000 (0.0111269%)
Fragment  Initial tries  Resolve tries  Successes   Failures
1         3000           0              3000        0

eq merge([], OL:SortedList{NatAsToset}) = OL:SortedList{
  NatAsToset} .
rewrites: 3000 (0.0111269%)

ceq merge(N : OL:SortedList{NatAsToset}, M : OL':SortedList{
  NatAsToset}) = N :merge(OL:SortedList{NatAsToset}, M :
```

```
  OL':SortedList{NatAsToset}) if N <= M = true .
lhs matches: 18705    rewrites: 18705 (0.0693761%)
Fragment  Initial tries  Resolve tries  Successes   Failures
1         18705          0              18705       0
```

5. The information for the `quicksort` algorithm follows a similar pattern as well. However, in this case it is interesting to notice that the equations for the `leq-elems` and `gr-elems` operations are attempted the same number of times, and for each of these operations the condition (say, `N <= M`) of one of the equations fails in around half of the cases, being then used the other equation (with condition, say, `N > M`.

```
eq quicksort([]) = [] .
rewrites: 3006 (0.0111491%)

eq quicksort(N : L:List{TOSET}{NatAsToset}) = quicksort(
  leq-elems(L:List{TOSET}{NatAsToset}, N)) ++ N : quicksort(
  gr-elems(L:List{TOSET}{NatAsToset}, N)) .
rewrites: 3003 (0.011138%)

eq leq-elems([], M) = [] .
rewrites: 3003 (0.011138%)

ceq leq-elems(N : L:List{TOSET}{NatAsToset}, M) = N :
  leq-elems(L:List{TOSET}{NatAsToset}, M) if N <= M = true .
lhs matches: 1012626  rewrites: 506277 (1.87776%)
Fragment  Initial tries  Resolve tries  Successes   Failures
1         1012626        0              506277      506349

ceq leq-elems(N : L:List{TOSET}{NatAsToset}, M) = leq-elems(
  L:List{TOSET}{NatAsToset}, M) if N > M = true .
lhs matches: 506349   rewrites: 506349 (1.87803%)
Fragment  Initial tries  Resolve tries  Successes   Failures
1         506349         0              506349      0

eq gr-elems([], M) = [] .
rewrites: 3003 (0.011138%)

ceq gr-elems(N : L:List{TOSET}{NatAsToset}, M) = gr-elems(
  L:List{TOSET}{NatAsToset}, M) if N <= M = true .
lhs matches: 1012626  rewrites: 506277 (1.87776%)
Fragment  Initial tries  Resolve tries  Successes   Failures
1         1012626        0              506277      506349

ceq gr-elems(N : L:List{TOSET}{NatAsToset}, M) = N : gr-elems(
  L:List{TOSET}{NatAsToset}, M) if N > M = true .
lhs matches: 506349   rewrites: 506349 (1.87803%)
Fragment  Initial tries  Resolve tries  Successes   Failures
1         506349         0              506349      0
```

6. From the rest of the equations applied we may highlight those for the `head` and `_++_` operations.

```
eq head(E:Nat : L:List{TOSET}{NatAsToset}) = E:Nat .
rewrites: 4795972 (17.7881%)
```

```
eq [] ++ L:List{TOSET}{NatAsToset} = L:List{TOSET}{
  NatAsToset} .
rewrites: 12006 (0.0445298%)

eq (E:Nat : L:List{TOSET}{NatAsToset}) ++ L':List{TOSET}{
  NatAsToset} =E:Nat : (L:List{TOSET}{NatAsToset} ++ L':List{
  Toset}{NatAsToset}) .
rewrites: 5010777 (18.5848%)
```

The equation for the `head` function is used in the evaluation of the condition of the membership axiom. The concatenation operator is used in the `quicksort`, `nats-upto`, and `reverse` functions.

Taking all the information provided by the profiler into account, we may think of doing different types of modifications to the original specification.

- None of the operators seems to be appropriate for memoization, since they are used on many different arguments, and if repeated, the size of the argument lists is so big that it is probably not worthy storing the results.

  Let us, in any case, add the `memo` attribute, e.g., to the `head` operator; the result for one of the reductions above is the following:

  ```
  Maude> red insertion-sort(random-nats(1000)) .
  rewrites: 1535372 in 18500ms cpu (19221ms real) (82992 rews/sec)
  result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...
  ```

  That is, it goes from 1286 milliseconds cpu time to 18500. See Figure 20.1 for the rest of the values.

- Conditional equations are in general computationally expensive. Let us write the two conditional equations for `insertion-sort` as one single unconditional equation:

  ```
  eq insert-list(N : OL, M)
    = if M <= N
      then M : N : OL
      else N : insert-list(OL, M)
      fi .
  ```

  The result for the same reduction is the following:

  ```
  Maude> reduce insertion-sort(random-nats(1000)) .
  rewrites: 1536365 in 638ms cpu (704ms real) (2404692 rews/sec)
  result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...
  ```

  Although the number of rewrites increases slightly—from 1535372 to 1536365—, the amount of cpu time has dropped to a half—from 1286 ms. to 638 ms.

- We may use the `owise` attribute for making the conditional equation of the `mergesort` function unconditional, that is, we may write it as:

  ```
  eq mergesort(L)
    = merge(mergesort(take (length(L) quo 2) from L),
            mergesort(throw (length(L) quo 2) from L))
    [owise] .
  ```

Although this clearly improves the equation, the win is not very significant. To test it, we run the `mergesort` algorithm on a randomly generated list.

```
Maude> red mergesort(random-nats(1000)) .
rewrites: 87005 in 68ms cpu (110ms real) (1261124 rews/sec)
result NeSortedList{NatAsToset}: 237728 : 17386481 : ...
```

The number of rewrites goes from 88005 to 87005 and the cpu time goes from 71 ms. to 68 ms.

- A more important gain may be obtained by improving the splitting of the lists for the quicksort algorithm. Let us join the `leq-elems` and `gr-elems` functions in one single `leq-gr-elems` returning a pair of lists, one with the smaller or equal elements and the other with the greater ones.

```
op quicksort : List{TOSET}{X} -> SortedList{X} .
op leq-gr-elems : List{TOSET}{X} X$Elt -> LGResult .
op leq-gr-elems : List{TOSET}{X} List{TOSET}{X} List{TOSET}{X}
    X$Elt -> LGResult .

sort LGResult .
op {_,_} : List{TOSET}{X} List{TOSET}{X} -> LGResult .

eq quicksort([]) = [] .
ceq quicksort(N : L)
 = quicksort(L') ++ (N : quicksort(L''))
 if {L', L''} := leq-gr-elems(L, N).

eq leq-gr-elems(L, M) = leq-gr-elems(L, [], [], M) .
eq leq-gr-elems([], L, L', M) = {L, L'} .
eq leq-gr-elems(N : L, L', L'', M)
  = if N <= M
    then leq-gr-elems(L, N : L', L'', M)
    else leq-gr-elems(L, L', N : L'', M)
    fi .
```

The execution of the `quicksort` function on a list of randomly generated numbers takes now 69612 rewrites (against 97858) and does the reduction in 31 milliseconds (around 75 before).

```
Maude> red quicksort(random-nats(1000)) .
reduce in SORTED-LIST-TEST : quicksort(random-nats(1000)) .
rewrites: 69612 in 31ms cpu (72ms real) (2175714 rews/sec)
result NeSortedList{NatAsToset}: 237728 : 17386481 : ...
```

There is still room for improvement in this specification. For instance, some operations on lists can be made more efficient by means of *tail-recursive* definitions with accumulator arguments, in the style of the definitions shown in Section 7.13.1.

20.1.6 Performance note

Turning on tracing, break points, or profiling causes Maude to run much more slowly, because these options force execution through a slow path that performs extensive bookkeeping before and after each rewrite, membership application, and condition fragment check. Therefore,

		original spec.		`memo head`	unconditional `insert-list`
		profiler on	profiler off		
`nats-upto`	rews	2012009			2013009
	ms	1169	1032	96188	1029
`reverse-nats-upto`	rews	5519515			5519515
	ms	4189	3634	193977	2376
`random-nats`	rews	1535372			1536365
	ms	1434	1286	18500	638

`insertion-sort`

		original spec.		`memo head`	`merge owise`
		profiler on	profiler off		
`nats-upto`	rews	2082358			2081358
	ms	1223	1079	95760	1079
`reverse-nats-upto`	rews	2578581			2577581
	ms	1409	1210	99579	1204
`random-nats`	rews	88005			87005
	ms	108	71	559	68

`mergesort`

		original spec.		`memo head`	improved splitting
		profiler on	profiler off		
`nats-upto`	rews	6519514			5267013
	ms	5671	5065	221387	2523
`reverse-nats-upto`	rews	6528518			5777017
	ms	4451	4096	196991	2650
`random-nats`	rews	97858			69612
	ms	83	75	693	31

`quicksort`

Figure 20.1: Number of rewrites and CPU time for different versions of the sorting algorithms

execution will be significantly slower if one or more of these options is on, even if no tracing information is output, no break points are encountered, and the profile information is never examined.

To ensure Maude is running at full speed one must use:

```
set trace off .
set break off .
set profile off .
```

20.2 Debugging strategy executions

The approaches described in Section 20.1 can also be applied for debugging the execution of strategies. When using the `srewrite` and `dsrewrite` commands, we can examine the equations, membership axioms, and rules that are executed as part of the strategy-controlled rewriting. Equations and membership axioms do not only occur in the evaluation of rules and rule conditions, but they can also appear in strategy-specific contexts like the evaluation of the conditions of the strategy operators, or the reduction of strategy call arguments and subtitution values. Moreover, we can also trace, stop, and profile each time a strategy definition is used to execute a strategy call.

However, debugging strategies has an additional hardness. Since the evaluation of strategies is a search for solutions in a restrained rewriting subtree (see Section 10.4), several execution paths may be being explored at the same time, and their traces will appear interleaved in the output. This affect `srewrite` more than `dsrewrite`.

Let us illustrate this with the Prolog example in Section 10.5 using the following logic program:

```
eq hypotheses := 'mortal(x{1}) :- 'fish(x{1}) ;
                 'mortal(x{1}) :- 'human(x{1}) ;
                 'fish(x{1}) :- 'cod(x{1}) ;
                 'human('Socrates) :- nil .
```

Tracing of strategy definitions is enabled by default, although it can be switched with the command

```
set trace sd on/off .
```

The trace of a strategy call includes the definition being applied, the call term (except for strategies without arguments), and the matching substitution for the left-hand side and the condition. The subject term being rewritten can also be shown by setting

```
set trace whole on .
```

In the example, we will disable tracing for any other statement, and execute the `srewrite` command using the `wsolve-simple` strategy, which recursively applies itself until a solution to the logic programming problem is found.

```
Maude> set trace mb off . set trace eq off . set trace rl off . set trace whole on .
Maude> srew < 'mortal('Socrates) | hypotheses > using wsolve-simple .
********** strategy call
sd wsolve-simple := ...
subject --> < 0 | 'mortal('Socrates) $ empty | (hypotheses) >
empty substitution
********** strategy call
sd solve-simple := ...
```

```
subject --> < 0 | 'mortal('Socrates) $ empty | (hypotheses) >
empty substitution
********** strategy call
sd solve-simple := ...
subject --> < 1 | 'fish(x{1}) $ x{1} -> 'Socrates | (hypotheses) >
empty substitution
********** strategy call
sd solve-simple := ...
subject --> < 1 | 'human(x{1}) $ x{1} -> 'Socrates | (hypotheses) >
empty substitution
********** strategy call
sd solve-simple := ...
subject --> < 2 | 'cod(x{2}) $ x{1} -> 'Socrates ; x{2} -> 'Socrates | (hypotheses) >
empty substitution
********** strategy call
sd solve-simple := ...
subject --> < 1 | nil $ x{1} -> 'Socrates | (hypotheses) >
empty substitution

Solution 1
rewrites: 256
result Configuration: solution(empty)

No more solutions.
rewrites: 256
```

We can observe the different proof paths are interleaved in the traces. The successful deduction *mortal* ← *human* ← *Socrates* is being explored at the same time as the failed deduction *mortal* ← *fish* ← *cod*. As we have said before, the `dsrewrite` command provides clearer traces, since the execution tree is explored in depth. However, to obtain the path to a given solution, we still have to discard failed branches that may have appeared during the search.

```
Maude> dsrew < 'mortal('Socrates) | hypotheses > using wsolve-simple .
********** strategy call
sd wsolve-simple := ...
subject --> < 0 | 'mortal('Socrates) $ empty | hypotheses >
empty substitution
********** strategy call
sd solve-simple := ...
subject --> < 0 | 'mortal('Socrates) $ empty | hypotheses >
empty substitution
********** strategy call
sd solve-simple := ...
subject --> < 1 | 'fish(x{1}) $ x{1} -> 'Socrates | hypotheses >
empty substitution
********** strategy call
sd solve-simple := ...
subject --> < 2 | 'cod(x{2}) $ x{1} -> 'Socrates ; x{2} -> 'Socrates | hypotheses >
empty substitution
********** strategy call
sd solve-simple := ...
subject --> < 1 | 'human(x{1}) $ x{1} -> 'Socrates | hypotheses >
empty substitution
********** strategy call
sd solve-simple := ...
```

```
subject --> < 1 | nil $ x{1} -> 'Socrates | hypotheses >
empty substitution

Solution 1
rewrites: 256
result Configuration: solution(empty)

No more solutions.
rewrites: 256
```

Tracing strategy calls is more instructive when they have parameters or when they are conditional, since we can see the matching substitution and the condition trials. For instance, we can execute part of the previous deduction with the `wsolve` strategy from Section 10.5.2. `wsolve` uses a auxiliary strategy `clauseWalk` to iterate through the program clauses and try them in order. We will put a label to one of its definitions, and limit tracing to it and to the `clause` rule.

```
sd clauseWalk(P :- PL2 ; Pr) := (clause[Pr2 <- Pr] ; solve)
                                | clauseWalk(Pr) [label walk] .
```

Then, we can see the matching substitution for the left-hand side of this `clauseWalk` definition. All the clauses of the program are tried in order, and only the last one succeeds.

```
Maude> set trace select on . trace select walk clause .
Maude> srew < 'human('Socrates) | hypotheses > using wsolve .
********** strategy call
sd clauseWalk(P:Predicate :- PL2:CfPredicateList ; Pr:Program) := ... [label walk] .
call term --> clauseWalk(hypotheses)
P:Predicate --> 'mortal(x{1})
PL2:CfPredicateList --> 'fish(x{1})
Pr:Program --> 'mortal(x{1}) :- 'human(x{1}) ; ... ; 'human('Socrates) :- nil
********** strategy call
sd clauseWalk(P:Predicate :- PL2:CfPredicateList ; Pr:Program) := ... [label walk] .
call term --> clauseWalk('mortal(x{1}) :- 'human(x{1}) ; ... ;
                         'human('Socrates) :- nil)
P:Predicate --> 'mortal(x{1})
PL2:CfPredicateList --> 'human(x{1})
Pr:Program --> 'fish(x{1}) :- 'cod(x{1}) ; 'human('Socrates) :- nil
********** strategy call
sd clauseWalk(P:Predicate :- PL2:CfPredicateList ; Pr:Program) := ... [label walk] .
call term --> clauseWalk('fish(x{1}) :- 'cod(x{1}) ; 'human('Socrates) :- nil)
P:Predicate --> 'fish(x{1})
PL2:CfPredicateList --> 'cod(x{1})
Pr:Program --> 'human('Socrates) :- nil
********** strategy call
sd clauseWalk(P:Predicate :- PL2:CfPredicateList ; Pr:Program) := ... [label walk] .
call term --> clauseWalk('human('Socrates) :- nil)
P:Predicate --> 'human('Socrates)
PL2:CfPredicateList --> (nil).CfPredicateList
Pr:Program --> (nil).Program
********** rule
crl ... [label clause] .
N1:Nat --> 0
P1:Predicate --> 'human('Socrates)
PL1:PredicateList --> (nil).PredicateList
```

```
S1:Substitution --> (empty).Substitution
Pr1:Program --> (hypotheses)
P2:Predicate --> 'human('Socrates)
PL2:PredicateList --> (nil).CfPredicateList
Pr2:Program --> (nil).Program
P3:Predicate --> 'human('Socrates)
PL3:PredicateList --> (nil).CfPredicateList
S2:Substitution --> (empty).Substitution
N2:Nat --> 0

Solution 1
rewrites: 69
result Configuration: solution(empty)

No more solutions.
rewrites: 69
```

Debugging is also available in the strategy-related commands: `srewrite` and `dsrewrite` can be invoked with the `debug` keyword prefix to enter the debugger, and break points can be set like for other statements. For instance, we could follow the clause walk of the previous trace using the debugger.

```
Maude> set break on . select break walk .
Maude> srew < 'mortal('Socrates) | hypotheses > using wsolve .
break on labeled strategy definition:
sd clauseWalk(P:Predicate :- PL2:CfPredicateList ; Pr:Program) := ... [label walk] .
Debug(1)> where .
Current term is:
< 0 | 'human('Socrates) $ empty | ^\textit{hypotheses}^ >
which arose while executing a top level command.
Debug(1)> resume .
break on labeled strategy definition:
sd clauseWalk(P:Predicate :- PL2:CfPredicateList ; Pr:Program) :=
    clause[Pr2:Program <- Pr:Program] ; solve | clauseWalk(Pr:Program) [label walk] .
Debug(1)> abort .
```

(1) Setting artificial break points (or trace points) at any position of a complex strategy expression is possible and could be useful when debugging strategies. Suppose we have the strategy

(1) Perhaps this is not good advice.

```
(r1 ; r2) ? r1 : r2
```

and we want to know what it the subject term just after `r1` and before `r2` in the condition. We can declare a strategy `bp1` and define it as

```
sd bp1 := idle [label bp1] .
```

Then, we evaluate the strategy `(r1 ; bp1 ; r2) ? r1 : r2` breaking or tracing at `bp1`. Conditional breakpoint can even be established, depending on the variables from the context or on the subject term, by means of a conditional strategy with arguments invoked like

```
(r1 ; matchrew S by S using bp(S, V) ; r2) ? r1 : r2
```

for some variable `S` of the subject sort, and some variables `V` from the context.

Finally, strategy commands can be profiled too. As an example, we include part of the profile of the first example in this section.

```
Maude> set profile on .
Maude> srew < 'mortal('Socrates) | hypotheses > using solve .
...
Maude> show profile .
...
crl < N1:Nat | P1:Predicate,PL1:PredicateList $ S1:Substitution |
    Pr1:Program ; P2:Predicate :- PL2:PredicateList ; Pr2:Program > => <
    N2:Nat | PL3:PredicateList,PL1:PredicateList $ S2:Substitution |
    Pr1:Program ; P2:Predicate :- PL2:PredicateList ; Pr2:Program > if
    P3:Predicate :- PL3:PredicateList := rename(P2:Predicate :-
    PL2:PredicateList, N1:Nat) /\ S2:Substitution := unify(P1:Predicate,
    P3:Predicate, S1:Substitution) /\ N2:Nat := max(N1:Nat, last(
    P3:Predicate :- PL3:PredicateList)) [label clause] .
rewrites: 4 (1.47059%)

sd solve := match Conf:Configuration such that isSolution(
    Conf:Configuration) = true or-else matchrew Conf:Configuration such
    that < N:Nat | CfPL:CfPredicateList $ S:Substitution | Pr:Program > :=
    Conf:Configuration by Conf:Configuration using clauseWalk(Pr:Program) |
    matchrew Conf:Configuration such that < N:Nat | CfPL:CfPredicateList,!,
    PL:PredicateList $ S:Substitution | Pr:Program > := Conf:Configuration
    by Conf:Configuration using (cut{solveOne} ; solve) .
rewrites: 5 (1.83824%)

sd clauseWalk((nil).Program) := fail .
rewrites: 4 (1.47059%)

sd clauseWalk(P:Predicate :- PL2:CfPredicateList ; Pr:Program) := clause[
    Pr2:Program <- Pr:Program] ; solve | clauseWalk(Pr:Program) [label
    walk] .
rewrites: 16 (5.88235%)
```

20.3 Traps and known problems

We list some commonly encountered problems with Maude.

20.3.1 Associativity and idempotency

Remember that the attributes `assoc` and `idem` (see Section 4.4.1) cannot be used together in any combination of attributes, because the appropriate matching and normalization algorithms have not been developed yet.

This requirement is quietly enforced by ignoring the attribute `idem` where necessary.

Let us consider the following example, in which we wrongly declare an operator with the attributes `assoc` and `idem` appearing together.

```
fmod WRONG-NAT-SET is
  pr NAT .
  sort WNatSet .
  subsort Nat < WNatSet .
  op none : -> WNatSet [ctor] .
  op __ : WNatSet WNatSet -> WNatSet
       [ctor assoc comm idem id: none] .
```

```
endfm
```

When we reduce a term like, e.g., `4 4 5 2`, the duplication does not disappear, because Maude has ignored the idempotency attribute; the remaining attributes are applied as usual.

```
Maude> red 4 4 5 2 .
result WNatSet: 2 4 4 5
```

We can solve this by adding explicitly an idempotency equation, as we have seen, for example, in Section 7.13.2.

Combining `idem` with attributes other than `assoc` is all right. For example, the following module combines idempotency with commutativity.

```
fmod COMM-IDEM-EX is
  pr NAT .
  sort CI .
  subsort Nat < CI .
  op f : CI CI -> CI [ctor comm idem] .

  vars N M : Nat .
  var  C : CI .

  op g : CI -> Nat .
  eq g(f(N, M)) = 0 .
  eq g(C) = 1 [owise] .
endfm

Maude> red f(2, 2) .
result NzNat: 2

Maude> red f(2, 3) == f(3, 2) .
result Bool: true
```

Notice that in this module, because of matching modulo commutativity and idempotency, the first equation for `g` can be applied to a term such as, e.g., `g(2)`. For example, we have the following reductions:

```
Maude> red g(2) .
result Zero: 0

Maude> red g(f(2, f(2, 2))) .
result Zero: 0

Maude> red g(f(2, f(3, 4))) .
result NzNat: 1
```

20.3.2 Segmentation fault (core dumped)

This looks like a bug in Maude, but in fact it is a stack overflow (a real segmentation fault is caught and reported as an "internal error"). On a Unix box you can find out the current limit on your stack size with the (shell) command

```
limit stacksize
```

This is often set to 8192K by default, which is quite inappropriate for a highly recursive system like Maude. You can set the stack size to a larger value with, for example,

```
limit stacksize 100M
```

or remove the limit altogether with

```
unlimit stacksize
```

Note that stack overflows are reported as `Illegal instruction` on both PowerPC- and Intel-based Macs.

20.3.3 Bare variable lefthand sides

The use of a bare variable lefthand side for an equation, rule, or membership axiom may lead to unexpected nontermination. The recommended place to use them is in statements declared with the `nonexec` attribute, which are only going to be applied via a strategy language. Using them in membership axioms is seductive, but very tricky. For example:

```
subsort Prime < Nat .
var N : Nat .
cmb N : Prime if favoritePrimeTest(N) .
```

will end up with the membership axiom and `favoritePrimeTest` being applied to every reduced term of sort `Nat`, including those that arise during evaluation of `favoritePrimeTest(N)` with likely nontermination.

20.3.4 Operator overloading and associativity

The situation where two ad-hoc overloaded operators have the same kinds in their arities but different ones in their coarities causes a warning to be emitted, as already mentioned in Section 3.6. For example, loading the file `overloading-assoc-warning.maude` containing the module

```
fmod OVER-ASSOC-EX1 is
  sorts Foo Bar .
  op f : Foo -> Foo .
  op f : Foo -> Bar .
endfm
```

causes the following warning:

```
Warning: "overloading-assoc-warning.maude", line 8 (fmod
  OVER-ASSOC-EX1): declaration for f has the same domain kinds as
  the declaration on "overloading-assoc-warning.maude", line 7
  (fmod OVER-ASSOC-EX1) but a different range kind.
```

A similar warning is obtained in the case where the arities differ but might look the same because of associativity, like in the following example (loaded as before):

```
fmod OVER-ASSOC-EX2 is
  sort Foo .
  op f : Foo Foo -> Foo [assoc] .
  op f : Foo Foo Foo -> Foo .
endfm

Warning: "overloading-assoc-warning.maude", line 22
  (fmod OVER-ASSOC-EX2): declaration for f clashes with
  declaration on "overloading-assoc-warning.maude", line 21 (fmod
  OVER-ASSOC-EX2) because of associativity.
```

20.3.5 Preregularity and equational attributes

We recall from Section 3.8 that Maude assumes that modules are *preregular* and generates warnings when a module contains operator declarations that do not satisfy this property. This means that for each possible combination of argument sorts the resulting term has a unique least type, which is usually a sort but might also be the kind, depending on the operator declarations. However, as also explained in Section 3.8, in the presence of equational attributes, such as `assoc`, `comm`, `id:`, and `idem` (see Section 4.4.1), preregularity must be understood *modulo* the axioms A declared by such attributes. That is, we want not just each term t, but also each equivalence class $[t]_A$ to have a least sort. Therefore, there is an additional requirement for an operator that is declared associative, namely, that the least type of a term should not depend on the way nested operators are associated. Let us explain this situation in some detail.

The `assoc` attribute, stating that a binary operator is associative, appears usually associated with declarations of operators whose arguments are both of the same sort, like, for example,

```
op _+_ : Nat Nat -> Nat [assoc] .
```

However, in the presence of subsorts and overloaded operators it also makes sense to have binary operators whose arguments are not the same, but are related via subsorting; for example, to make it explicit that the addition of a natural number to a nonzero natural number produces a nonzero natural number, we can have an additional declaration

```
op _+_ : NzNat Nat -> NzNat [assoc] .
```

or also (see Section 4.4.6)

```
op _+_ : NzNat Nat -> NzNat [ditto] .
```

Thus, in general, the `assoc` attribute is allowed for binary operators such that the two argument sorts and the result sort all belong to the same connected component. Therefore, it is possible to consider a module like the following:

```
fmod NON-ASSOCIATIVE-EX is
  sorts s1 s2 .
  subsort s1 < s2 .
  op f : s1 s2 -> s2 [assoc] .
  op a : -> s1 .
  eq f(a,a) = a .
endfm
```

If we try to reduce the term `f(a,a)`, we get the following warning:

```
Maude> red f(a, a) .
Warning: sort declarations for associative operator f are
  non-associative on 2 out of 27 sort triples. First such triple is
  (s1, s1, s2).
reduce in NON-ASSOCIATIVE-EX : f(a, a) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result s1: a
```

Maude has checked the preregularity property on the associative operator `f`. It is enough to check this property on each triple of types, and when it fails to hold Maude returns the first such triple. In this example we have three possible types for each one of the two arguments and also for the result, namely, the sorts `s1` and `s2`, and the corresponding kind `[s2]`, and therefore we have $3^3 = 27$ possible triples. Among those, the triple `(s1, s1, s2)` does *not* satisfy the preregularity checking, because `f(X:s1, X:s1)` has sort `s2`, `f(X:s1, X:s2)` has sort `s2`, and `f(X:s2, X:s2)` has kind `[s2]`, but no sort; thus the flattened term `f(X:s1, X:s1, X:s2)`

could have either sort `s2`, by grouping the arguments as `f(X:s1, f(X:s1, X:s2))`, or kind `[s2]`, by grouping the arguments as `f(f(X:s1, X:s1), X:s2))`. To sum up, the sort structure for the operator `f` is said to be *non-associative* on the triple `(s1, s1, s2)`.

Two ways of avoiding this undesirable situation are the following: either having a unique declaration at the top sort with both arguments of the same sort,

```
fmod ASSOCIATIVE-EX1 is
  sorts s1 s2 .
  subsort s1 < s2 .
  op f : s2 s2 -> s2 [assoc] .
  op a : -> s1 .
  eq f(a,a) = a .
endfm

Maude> red f(a, a) .
reduce in ASSOCIATIVE-EX1 : f(a, a) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result s1: a
```

or adding enough declarations to cover all possible combinations of arguments; in this case only one more declaration is enough, as follows:

```
fmod ASSOCIATIVE-EX2 is
  sorts s1 s2 .
  subsort s1 < s2 .
  op f : s2 s2 -> s2 [assoc] .
  op f : s1 s2 -> s2 [assoc] .
  op a : -> s1 .
  eq f(a,a) = a .
endfm

Maude> red f(a, a) .
reduce in ASSOCIATIVE-EX2 : f(a, a) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result s1: a
```

When an associative operator is also declared to be commutative using the `comm` atribute, Maude computes the commutative completion (switching the order of the argument sorts) of the given operator declarations before checking preregularity.

In the case of the `id:` and `idem` attributes, preregularity modulo those axioms requires that all *collapses*, that is, all passages from a term $f(t, e)$ to an equivalent term t by application of an identity equation $f(x, e) = x$; or from a term $f(t, t)$ to an equivalent term t by application of an idempotency equation $f(x, x) = x$ (where in both cases the top function symbol f disappears), should be such that the least sort of the resulting term t is smaller than or equal to the least sort of $f(t, e)$ (resp. $f(t, t)$). A term that can collapse from one sort to a greater or incomparable sort breaks the sort calculations and violates preregularity modulo such axioms. Therefore, syntactic conditions ensuring that a collapse is also into a lesser or equal sort are checked by Maude for both the `id:` and `idem` attributes.

20.3.6 Collapse theories

Using `id:` or `idem` attributes means that you are (conceptually) working with infinite equivalence classes, and that many lefthand side patterns will match in unexpected ways. Unlike OBJ3, Maude has true collapse matching algorithms, rather than identity completion, and it does not try to omit problematic matches. Consider for example the module

```
fmod COLLAPSE-ID-EX is
  sort Foo .
  ops a e : -> Foo .
  op f : Foo Foo -> Foo [left id: e] .
  var X : Foo .
  eq f(X, a) = ...
endfm
```

Then we have

```
a = f(e, a) = f(e, f(e, a)) = ...
```

In particular, the pattern `f(X, a)` matches `a` with `X` ← `e`, leading to possible nontermination. You should be wary of having an operator with an identity element as the top symbol for a lefthand side. One useful trick when you need a pattern like `f(X, a)` is to use a pattern `f(Y, a)` where `Y` has a sort lower than that of the identity element. For example,

```
fmod COLLAPSE-NAT-EX is
  sorts Nat NzNat .
  subsort NzNat < Nat .
  op 0 : -> Nat .
  op s : Nat -> NzNat .
  op + : Nat Nat -> Nat [assoc comm id: 0] .
  op + : Nat NzNat -> Nat [assoc comm id: 0] .
  var X : Nat .
  var Y : NzNat .
  eq +(s(X), Y) = s(+(X, Y)) .
endfm
```

Here `+(s(X), Y)` cannot match `s(0)` because, although `s(0) = +(s(0), 0)` by the identity attribute, `Y` cannot match `0`.

Rewriting with the `idem` attribute is even riskier. For example,

```
fmod COLLAPSE-IDEM-EX is
  sort Foo .
  ops a b : -> Foo .
  op f : Foo Foo -> Foo [idem] .
  var X : Foo .
  eq a = b .
endfm
```

We then have

```
a = f(a, a) = f(f(a, a), f(a, a)) = ...
```

Thus, if `a` can be rewritten by an equation, then any number of rewrites can be done by using the `idem` axiom to create new copies of `a`. In fact, the current implementation would choose the obvious rewrite and just produce `b`, but this should not be relied upon; `COLLAPSE-IDEM-EX` is a nonterminating system. The only safe way to use `idem` is as follows. Whenever a connected component is the domain and range of an operator having the `idem` attribute, then its sorts are *poisoned*. Terms of poisoned sorts must never rewrite other than by rules under the control of a strategy, that is, using metalevel descent functions. Such terms must be built out of free constructors—operators that may have equational attributes such as `comm`, but may not have equations with these operators at the top. Of course, it is ok to have defined functions that work on such constructor terms; it is just that the terms themselves may not rewrite.

20.3.7 One-sided identities and associativity

When the associativity axiom is combined with a one-sided identity axiom some unexpected matching properties result. Consider the module:

```
fmod ASSOC-ID-EX is
  sort Foo .
  ops a b 1f : -> Foo .
  op f : Foo Foo -> Foo [assoc left id: 1f] .
  var X Y : Foo .
endfm
```

Then (see Section 23.3 for matching commands),

```
match f(X, Y) <=? f(a, b) .
```

yields three solutions:

```
Solution 1
X:Foo --> 1f
Y:Foo --> f(a, b)

Solution 2
X:Foo --> a
Y:Foo --> b

Solution 3
X:Foo --> f(a, 1f)
Y:Foo --> b
```

whereas the naive user may not have expected the last solution.

Matching with extension can be even more surprising. The command

```
xmatch f(X, Y) <=? f(a, b) .
```

yields five solutions:

```
Solution 1
Matched portion = f(a, 1f)
X:Foo --> a
Y:Foo --> 1f

Solution 2
Matched portion = f(a, 1f)
X:Foo --> f(a, 1f)
Y:Foo --> 1f

Solution 3
Matched portion = (whole)
X:Foo --> 1f
Y:Foo --> f(a, b)

Solution 4
Matched portion = (whole)
X:Foo --> a
Y:Foo --> b
```

```
Solution 5
Matched portion = (whole)
X:Foo --> f(a, 1f)
Y:Foo --> b
```

Here the first two solutions match a portion `f(a, 1f)` of the subject that was not apparent from the original problem. However, if one considers the equivalence class of `f(a, b)` they are valid solutions that are necessary for correct simulation of (conditional) rewriting on equivalence classes.

20.3.8 Memberships for associative operators

Membership axioms can interact with `assoc` or `iter` operator attributes in undesirable ways.

The reason is that, for completeness, the operator declarations would have to be tried on every subterm of every member of the equivalence class, and this is not done (for efficiency reasons) in the current implementation, giving rise to some warnings.

For associative operators declared at the sort level, membership axioms will be applied only at the top, they will not be applied to subterms in the process of applying an operator declaration to compute the sort. For example in the following module

```
fmod ASSOC-MB-EX1 is
  sort Foo .
  op f : Foo Foo -> Foo [assoc comm] .
  op e : -> [Foo] .
  ops a b c d : -> Foo .

  mb f(a, e) : Foo .
endfm
```

the membership axiom will not be used to lower the sort of `f(a, f(b, e))` to `foo` as it does not match at the top.

Recall from Sections 3.9.3 and 4.8 that terms built with associative operators can be written in flattened form. This is the notation used for `f`-terms in the following examples.

```
Maude> red f(a, b, e) .
Warning: membership axioms are not guaranteed to work correctly for
    associative symbol f as it has declarations that are not at the
    kind level.
reduce in ASSOC-MB-EX1 : f(e, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, a, b)

Maude> red f(a, b, e, a) .
reduce in ASSOC-MB-EX1 : f(e, a, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, a, a, b)

Maude> red f(e, b, e, a) .
reduce in ASSOC-MB-EX1 : f(e, e, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, e, a, b)

Maude> red f(a, b, e, e, a) .
reduce in ASSOC-MB-EX1 : f(e, e, a, a, b) .
```

```
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, e, a, a, b)

Maude> red f(a, a, b, e, e, a) .
reduce in ASSOC-MB-EX1 : f(e, e, a, a, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, e, a, a, a, b)
```

Here the intuition is that each `e` forces the result to the kind level, unless there is an `a` to bring it back down. Unfortunately, for `f(a, b, e)` we would need to use the membership axiom on a proper subterm, and then use the declaration at the top to arrive at the sort `Foo`, and this is not allowed.

Note that the warning produced by Maude is a per module warning and is only printed once, when the first reduction or rewriting command is given in the module.

The module `ASSOC-MB-EX1` above can be rewritten so that sort computations work as expected as follows:

```
fmod ASSOC-MB-EX2 is
  sort Foo .
  op f : [Foo] [Foo] -> [Foo] [assoc comm] .
  op e : -> [Foo] .
  ops a b c d : -> Foo .

  mb f(X:Foo, Y:Foo) : Foo .
  mb f(a, e) : Foo .
endfm

Maude> red f(a, b, e) .
reduce in ASSOC-MB-EX2 : f(e, a, b) .
rewrites: 2 in 0ms cpu (1ms real) (~ rews/sec)
result Foo: f(e, a, b)

Maude> red f(a, b, e, a) .
reduce in ASSOC-MB-EX2 : f(e, a, a, b) .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e, a, a, b)

Maude> red f(e, b, e, a) .
reduce in ASSOC-MB-EX2 : f(e, e, a, b) .
rewrites: 6 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, e, a, b)

Maude> red f(a, b, e, e, a) .
reduce in ASSOC-MB-EX2 : f(e, e, a, a, b) .
rewrites: 11 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e, e, a, a, b)

Maude> red f(a, a, b, e, e, a) .
reduce in ASSOC-MB-EX2 : f(e, e, a, a, a, b) .
rewrites: 12 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e, e, a, a, a, b)
```

Here the operator declaration is at the *kind level*, and the effect of the declaration of `f` in `ASSOC-MB-EX1` is obtained by an extra membership axiom.[4]

Let us see another example of this situation, starting with a module specifying non-empty lists of natural numbers.

```
fmod SIMPLE-NAT-LIST is
  protecting NAT .
  sort NatList .
  subsort Nat < NatList .
  op __ : NatList NatList -> NatList [assoc] .
endfm
```

It seems natural to specify *sorted* lists of natural numbers by importing `SIMPLE-NAT-LIST` and then defining a subsort of `NatList`.

```
fmod NAIVE-SORTED-NAT-LIST is
  protecting SIMPLE-NAT-LIST .
  sort SortedNatList .
  subsort Nat < SortedNatList < NatList .

  vars I J : Nat .
  var  SNL : SortedNatList .
  cmb I J : SortedNatList if I <= J .
  cmb I J SNL : SortedNatList if I <= J /\ J SNL : SortedNatList .
endfm

Maude> red 0 1 2 3 4 5 6 7 8 9 .
Warning: membership axioms are not guaranteed to work correctly for
    associative symbol __ as it has declarations that are not at the
    kind level.
reduce in NAIVE-SORTED-NAT-LIST : 0 1 2 3 4 5 6 7 8 9 .
rewrites: 1354 in 0ms cpu (0ms real) (~ rews/sec)
result SortedNatList: 0 1 2 3 4 5 6 7 8 9
```

To avoid this, we can rewrite the module above so that we only use *kind-level operator declarations* (notice the form of the arrow) and convert all sort-level operator declarations into memberships.

```
fmod NAT-LIST-KIND is
  protecting NAT .
  sort NatList .
  subsort Nat < NatList .

  op __ : NatList NatList ~> NatList [assoc] .
  mb I:NatList J:NatList : NatList .
endfm

fmod SORTED-NAT-LIST-KIND is
  protecting NAT-LIST-KIND .
  sort SortedNatList .
  subsort Nat < SortedNatList < NatList .
```

[4]Maude 1 did not allow multiple membership axioms on associative operators. In Maude 2 this works, although it will be extremely inefficient for large terms, since matching the extra membership essentially amounts to expanding out the equivalence class.

```
  vars I J : Nat .
  var  SNL : SortedNatList .
  cmb I J : SortedNatList if I <= J .
  cmb I J SNL : SortedNatList if I <= J /\ J SNL : SortedNatList .
endfm

Maude> red 0 1 2 3 4 5 6 7 8 9 .
reduce in SORTED-NAT-LIST-KIND : 0 1 2 3 4 5 6 7 8 9 .
rewrites: 1354 in 0ms cpu (0ms real) (~ rews/sec)
result SortedNatList: 0 1 2 3 4 5 6 7 8 9
```

20.3.9 Memberships for iterated operators

In analogy to interaction of associative operators and membership declarations, terms constructed with a stack of iterated operators may not be assigned the expected sort when it is necessary to apply a membership axiom to a subterm in order to infer the sort. Again, if an `iter` operator is declared at the sort level, Maude will not apply membership axioms to subterms in order to calculate the sort of a subterm before attempting to apply the operator declaration to calculate the sort of the whole term. As an example, consider the following module:

```
fmod ITER-MB-EX1 is
  sort Foo .
  op f : Foo -> Foo [iter] .
  op e : -> [Foo] .

  mb f(e) : Foo .
endfm

Maude> red f(e) .
Warning: membership axioms are not guaranteed to work correctly for
    iterated symbol f as it has declarations that are not at the
    kind level.
reduce in ITER-MB-EX1 : f(e) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e)

Maude> red f(f(e)) .
reduce in ITER-MB-EX1 : f^2(e) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f^2(e)

Maude> red f(f(f(e))) .
reduce in ITER-MB-EX1 : f^3(e) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f^3(e)
```

Here the intuition is that `e` is at the kind level, but `f(e)` is not. Unfortunately, for `f(f(e))` we would need to use the membership axiom on a proper subterm and then use the declaration at the top to arrive at the sort `Foo`, and declarations applying above membership axioms for iterated operators are not allowed.

Again, recall that the warning that membership axioms may not work is only given once per module. Here it just happens that it is given in response to a reduction command that does give the right answer.

The example can be rewritten so that membership axioms can be used to compute the desired sort as follows:

```
fmod ITER-MB-EX2 is
  sort Foo .
  op f : [Foo] -> [Foo] [iter] .
  op e : -> [Foo] .

  mb f(X:Foo) : Foo .
  mb f(e) : Foo .
endfm

Maude> red f(e) .
reduce in ITER-MB-EX2 : f(e) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e)

Maude> red f(f(e)) .
reduce in ITER-MB-EX2 : f^2(e) .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f^2(e)

Maude> red f(f(f(e))) .
reduce in ITER-MB-EX2 : f^3(e) .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f^3(e)
```

Here the operator declaration is at the *kind level*, and as in the associativity example in the previous section, the effect of the old declaration is obtained by an extra membership axiom. Note that using membership axioms in this way loses the efficiency for big towers of operators, which is the whole point of the `iter` theory.

20.3.10 Ambiguity in `print` attribute items

Since Maude has few restrictions on variable names, it is possible to introduce ambiguity with the `print` attribute by using strings or attribute names as variables. Consider, for example, the following module, where the string `"here"` and the keyword `metadata` are also declared as variables.

```
fmod PRINT-ATTR-AMBIGUOUS is
  sort Foo .
  op a : -> Foo .
  ops f g h : Foo -> Foo .
  vars metadata "here" : Foo .
  eq f("here") = g("here") [print "here"] .
  eq g(metadata) = h(metadata)
     [print "metadata = " metadata "g->h equation"] .
endfm
```

In the `print` attribute of the first equation, Maude cannot decide whether `"here"` is a string constant or a variable. Similarly, in the `print` attribute of the second equation, Maude will not be able to decide whether `metadata` is a variable or a keyword. Under these circumstances, Maude will output warnings about the multiple parses and then make an undefined choice between them.

```
Warning: <standard input>, line 6 (fmod PRINT-ATTR-AMBIGUOUS):
  multiple distinct parses for statement
  eq f ("here") = g ("here") [print "here"] .
Warning: <standard input>, line 7 (fmod PRINT-ATTR-AMBIGUOUS):
  multiple distinct parses for statement
  eq g (metadata) = h (metadata)
     [print "metadata = " metadata "g->h equation"] .
```

In this particular example, the equations work but, as a consequence of the ambiguity, the user does not get the expected information provided by the `print` attribute.

```
Maude> set print attribute on .
Maude> red f(g(a)) .
reduce in PRINT-ATTR-AMBIGUOUS : f(g(a)) .
metadata =
h(a)
metadata =
rewrites: 3 in 0ms cpu (0ms real) (25210 rewrites/second)
result Foo: h(h(a))
```

Part II

Full Maude

Chapter 21

Full Maude: Extending Core Maude

During the development of the Maude system we have put special emphasis on the creation of metaprogramming facilities to allow the generation of execution environments for a wide variety of languages and logics. The first most obvious area where Maude can be used as a metalanguage is in building language extensions for Maude itself. Our experience in this regard—first reported in [49], and further documented in [50, 44, 45, 51]—is very encouraging.

We have been able to define in Core Maude a language, that we call Full Maude, with all the features of Maude plus notation for object-oriented programming, module expressions specifying tuples of any size, etc. Although the Maude distribution has included the specification/implementation of Full Maude since it was first distributed in 1999, Core Maude and Full Maude are now closer than ever before. Many of the features now available in Core Maude, like the strategy language, unification and narrowing, parameterized modules and views, and module expressions like summation, renaming and instantiation, were available in Full Maude long before they were available in Core Maude [49]. In fact, Full Maude has not only been a complement to Core Maude, but also a vehicle to experiment with new language features. Once these features have been mature enough to be implemented in the core language, we have made the effort to do so. Similarly, it is very likely that those features in Full Maude which are not yet available in Core Maude will become part of it sooner or later, and that new features will be added to Full Maude for purposes of language design and experimentation. This applies not only to Full Maude, but also to further language extensions based on Full Maude such as the strategy language proposed in [98], whose Core Maude implementation is currently underway.

Full Maude implements a complete user interface for the extended language. Using the `META-LEVEL` and other predefined modules, we have been able to define in Core Maude all the additional functionality required for parsing, evaluating, and pretty-printing modules in the extended language, and also for input/output interaction, as already discussed in Chapter 18. Thanks to the efficient implementation of the rewrite engine, the parser, and the `META-LEVEL` module, such a language extension executes with reasonable efficiency.

Full Maude contains Core Maude as a sublanguage, so that Core Maude modules can also be entered at the Full Maude level. However, currently there are a few syntactic restrictions that have to be satisfied by modules and commands in order to be acceptable inputs at the Full Maude level, including the fact that Full Maude inputs, for both modules and commands, must be enclosed in parentheses. These syntactic restrictions are explained in Section 21.6.

The structure of this chapter is as follows. Section 21.1 gives instructions on how to load

and use Full Maude, how to enter modules, reduce terms, trace executions, etc. Section 21.2 explains how modules in Core Maude's database may be used from Full Maude. Section 21.3 introduces the additional module operations that are available in Full Maude. Section 21.4 explains how to move terms and modules up and down reflection levels. Finally, Section 21.6 enumerates the main differences between Full Maude and Core Maude.

21.1 Running Full Maude

Since the execution environment for Full Maude has been implemented in Core Maude, to initialize the system so that we can start using it the first thing we have to do is to load the `FULL-MAUDE` module in the system. Assuming that the file `full-maude.maude`, which contains the executable specification of Full Maude, is located in the current directory (or in a place where Maude can locate it, see Section 2.2), we just need to type the corresponding `in` or `load` command in the Maude prompt:

```
Maude> load full-maude.maude

         Full Maude 3.1 Oct 12 2020
```

The Full Maude system is then loaded, and we can use it as any other module.

Since Maude can take file names as arguments when started, assuming one is working on a Linux platform, one may also run Maude as follows:

```
~/maude-linux/bin$ ./maude.linux64 full-maude.maude
                    \||||||||||||||||||/
                  --- Welcome to Maude ---
                    /||||||||||||||||||\
            Maude 3.1 built: Oct 12 2020 20:12:31
            Copyright 1997-2020 SRI International
                 Tue Oct 13 12:00:00 2020

         Full Maude 3.1 Oct 12 2020
```

At the end of this file `full-maude.maude` there is the command

```
loop init .
```

which initializes the system just after loading the specification. This command starts the read-eval-print loop (see Section 18.4) to allow the interaction with the user by entering modules, theories, views, and commands, and to maintain a database in which to store all the modules, theories and views being introduced. The term `init` is a constant of sort `System`, in the specification of Full Maude, standing for the initial state of the Full Maude database.

Typing control-C may result in the loop being broken, and with it the current execution of Full Maude. Maude may try to recover the loop by itself, but if it is not successful we must reinitialize it with the `loop` command. That is, we need to type

```
Maude> loop init .
```

This command will be successful only if the `full-maude.maude` file is loaded and the `FULL-MAUDE` module is the default one. If it is not the default one, we may select it with the `select` command (see Section 23.15):

```
Maude> select FULL-MAUDE .
Maude> loop init .
```

The `loop init` command may be omitted here: Maude will try to restart the loop, using the last `loop` command, if something is written in parentheses henceforth.

Any module, theory, view, or command intended for Full Maude has to be enclosed in parentheses. Since Core Maude is running underneath Full Maude—indeed, it now provides what might be called the *system programming* level—it will handle any input not enclosed in parentheses. This allows the possibility of using both systems at the same time. Thanks to this, we may use many Core Maude commands when interacting with Full Maude. For example, we may use Core Maude trace or profile facilities on Full Maude specifications, may load files, etc. However, this may lead to some confusion, and we should take care of putting parentheses around those pieces of text intended for Full Maude.

A Core Maude module, such as those presented in previous sections, can be entered in Full Maude by enclosing it in parentheses. For example, a module `PATH`[1] can be entered to Full Maude as follows:

```
Maude> (fmod PATH is
         sorts Node Edge .
         ops source target : Edge -> Node .

         sort Path .
         subsort Edge < Path .
         op _;_ : [Path] [Path] -> [Path] .

         var  E : Edge .
         vars P Q R S : Path .
         cmb E ; P : Path if target(E) = source(P) .
         ceq (P ; Q) ; R
           = P ; (Q ; R)
           if target(P) = source(Q) /\ target(Q) = source(R) .

         ops source target : Path -> Node .
         ceq source(P) = source(E) if E ; S := P .
         ceq target(P) = target(S) if E ; S := P .

         protecting NAT .

         ops n1 n2 n3 n4 n5 : -> Node .
         ops a b c d e f : -> Edge .
         op length : Path -> Nat .

         eq length(E) = 1 .
         ceq length(E ; P) = 1 + length(P) if E ; P : Path .

         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 .
```

[1] Some fragments of this module have been discussed in Sections 3.5 and 4.3.

```
    eq source(f) = n2 .
    eq target(f) = n1 .
  endfm)

rewrites: 5438 in 10ms cpu (157ms real) (543800 rews/sec)
Introduced module PATH
```

As in Core Maude, we can enter any module or command by writing it directly after the prompt, or by having it in a file and then using the `in` or `load` commands. Also as in Core Maude, we can write several Full Maude modules or commands in a file and then enter all of them with a single `in` or `load` command (without parentheses), but each of the modules or commands has to be enclosed in parentheses.

When entering a module, as above, Maude gives us information about the rewrites executed to handle such a module. This is the number of rewrites done by Full Maude to evaluate the module being entered. In the same way, every time we enter a command, although in most cases it finally makes a call to Core Maude, Full Maude needs to perform some additional rewrites. Thus, as we will see below, the number of rewrites given by the system for Full Maude commands includes the reductions due to the evaluation of the command and those due to the execution of the command itself.

We can perform reduction or rewriting using a syntax for commands such as that of Core Maude.

```
Maude> (red in PATH : b ; (c ; d) .)
rewrites: 893 in 30ms cpu (21ms real) (29766 rewrites/second)
reduce in PATH :
  b ;(c ; d)
result Path :
  b ;(c ; d)

Maude> (red length(b ; (c ; d)) .)
rewrites: 474 in 10ms cpu (2ms real) (47400 rewrites/second)
reduce in PATH :
  length(b ;(c ; d))
result NzNat :
  3

Maude> (red a ; (b ; c) .)
rewrites: 587 in 0ms cpu (2ms real) (~ rewrites/second)
reduce in PATH :
  a ;(b ; c)
result [Path] :
  a ;(b ; c)

Maude> (red source(a ; (b ; c)) .)
rewrites: 616 in 0ms cpu (2ms real) (~ rewrites/second)
reduce in PATH :
  source(a ;(b ; c))
result [Node] :
  source(a ;(b ; c))

rewrites: 622 in 0ms cpu (2ms real) (~ rewrites/second)
reduce in PATH :
  target((a ; b); c)
result [Node] :
  target((a ; b); c)
```

```
Maude> (red length(a ; (b ; c)) .)
rewrites: 579 in 0ms cpu (2ms real) (~ rewrites/second)
reduce in PATH :
  length(a ;(b ; c))
result [Nat] :
  length(a ;(b ; c))
```

Note the number of rewrites. These figures include, as said above, the rewrites accomplished by Full Maude in the processing of the inputs and outputs, plus the number of rewrites of the reduction itself. For example, the first two reductions above in Core Maude would produce the following output:

```
Maude> red in PATH : b ; (c ; d) .
reduce in PATH : b ; (c ; d) .
rewrites: 7 in 0ms cpu (23ms real) (~ rews/sec)
result Path: b ; (c ; d)

Maude> red length(b ; (c ; d)) .
reduce in PATH : length(b ; (c ; d)) .
rewrites: 12 in 0ms cpu (0ms real) (~ rews/sec)
result NzNat: 3
```

Tracing, debugging, profiling, and the other facilities available in Core Maude (see Section 20.1) are also available in Full Maude. Since these facilities are provided by Core Maude, the corresponding commands for managing them must be written without parentheses. For example, we can do the following:

```
Maude> set trace on .
Maude> set trace mb off .
Maude> set trace condition off .
Maude> set trace substitution off .
Maude> (red length(b ; c) .)
********** trial #1
ceq length(E:Edge ; P:Path) = length(P:Path) + 1
  if E:Edge ; P:Path : Path .
********** solving condition fragment
E:Edge ; P:Path : Path
********** success for condition fragment
E:Edge ; P:Path : Path
********** success #1
********** equation
ceq length(E:Edge ; P:Path) = length(P:Path) + 1
  if E:Edge ; P:Path : Path .
length(b ; c)
--->
length(c) + 1
********** equation
eq length(E:Edge) = 1 .
length(c)
--->
1
********** equation
(built-in equation for symbol _+_)
1 + 1
--->
```

```
2
rewrites: 444 in 0ms cpu (7ms real) (~ rewrites/second)
reduce in PATH :
  length(b ; c)
result NzNat :
  2
```

One should always bear in mind that Full Maude is part of the specification being run. The specification of Full Maude is loaded in the system, and as said above, some of the rewrites taking place are the result of applying equations or rules in these modules. In the case of tracing, the rewrites done by Full Maude are not shown thanks to one of the trace commands available, namely `trace exclude` (see Sections 20.1.1 and 23.8). With such a command we may exclude particular modules from being traced. In particular, the `full-maude.maude` file includes the command `trace exclude FULL-MAUDE`, where `FULL-MAUDE` is the top module of the specification of Full Maude.

21.2 Using Core Maude modules in Full Maude

Full Maude maintains a module database independent from the one used by Core Maude to store the modules entered into it. In fact, this module database is a Maude term stored as part of the state in the `LOOP-MODE` input/output object. Therefore, a module entered into Core Maude can only import modules previously entered into Core Maude. However, Full Maude modules can import modules previously entered either into Full Maude or into Core Maude. Basically, if Full Maude cannot find a module in its own database, it looks into Core Maude's module database to find it.

When metaprogramming, the system behaves differently. In Core Maude, a metamodule (that is, the metarepresentation of a module) can include a module at the object level. In Full Maude, however, metamodules cannot import modules entered into Full Maude and can only import modules entered into Core Maude. Note that Full Maude is implemented using reflection, and that in the end all modules are handled by Core Maude, which is not aware of Full Maude's database.

Notice also that loading a Core Maude module once Full Maude is running will break the read-eval-print loop (see Section 18.4). Therefore, one should enter such modules *before* starting Full Maude. Assuming there is a file `path.maude` containing the Core Maude module `PATH`, we will have the following behavior if we enter it into Full Maude.

```
Maude> load path.maude

Maude> (red in PATH : b ; (c ; d) .)
Warning: no loop state.
Advisory: attempting to reinitialize loop.
Warning: "full-maude.maude", line 13692: bad token init
Warning: "full-maude.maude", line 13692: no parse for term.
Advisory: unable to reinitialize loop.
```

As said above, when the loop gets broken, as in this case, we must select the `FULL-MAUDE` module and restart the loop. We may now do the following:

```
Maude> select FULL-MAUDE .
Maude> loop init .

         Full Maude 3.1 Oct 12 2020
```

```
Maude> (red in PATH : b ; (c ; d) .)
reduce in PATH :
  b ;(c ; d)
result Path :
  b ;(c ; d)
```

Notice that with a `loop init` command Full Maude is restarted with an empty database. That is, any Full Maude module entered before the reinitialization will have to be reentered again. In this case, `PATH` is a Core Maude module, which is being executed *in* Full Maude. Since it is not in Full Maude's database, Full Maude looks into Core Maude's database and then executes the command in it. This functionality is useful for using any of the predefined modules, but also other modules which are not part of Maude's prelude. For example, for using inside Full Maude the model checker, which although predefined is not part of the `prelude.maude` file, we just need to load the `model-checker.maude` file *before* starting the loop. For example, we can do the following:

```
~/maude-linux/bin$ ./maude.linux64 model-checker.maude full-maude.maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 3.1 built: Oct 12 2020 20:12:31
            Copyright 1997-2020 SRI International
                  Tue Oct 13 12:00:00 2020

                Full Maude 3.1 Oct 12 2020

Maude> (mod CHECK-RESP is
         protecting MODEL-CHECKER .
         ...
       endm)

Maude> (red p(0) |= (<> Qstate) .)
```

See Section 22.7 for a concrete example of the use of Maude's model checker with Full Maude modules.

21.3 Additional module operations in Full Maude

As for Core Maude, in Full Maude we can use the keywords `protecting`, `extending`, and `including` (or `pr`, `ex`, and `inc` in abbreviated form) to define structured specifications, as well as summation, renaming, and instantiation operations on parameterized modules (see Chapter 6). All the predefined modules introduced in Chapter 7, plus the module `META-LEVEL` and its submodules, described in Chapter 17, are also available in Full Maude.

In addition to the module operations available in Core Maude, Full Maude supports the following extensions:

- *Tuple* and *power expressions* which, given any nonzero natural number, generate parameterized modules specifying *tuples* and *powers* of the corresponding size.

  `TUPLE[`⟨*NonzeroNaturalNumber*⟩`]\{`⟨*ViewExpression*⟩`\}`
  `POWER[`⟨*NonzeroNaturalNumber*⟩`]\{`⟨*ViewExpression*⟩`\}`

  See Section 21.3.1.

- *Object-oriented modules*, extending all the module operations available in Core Maude to this new type of modules. Thus, in Full Maude we may rename object-oriented modules, with renamings of classes, attributes, and messages, or use object-oriented modules in the summation of modules. Full Maude also supports object-oriented theories, views from object-oriented theories to object-oriented modules, and object-oriented parameterized modules, as well as the instantiation of such object-oriented parameterized modules. We devote Chapter 22 to the study of object-oriented modules.

As in Core Maude, a module or theory importing some combination of modules or theories, given by module expressions, can be seen as a structured module with more or less complex relationships among its component submodules. For execution purposes, however, we typically want to convert this structured module into an equivalent unstructured module, that is, into a "flattened" module without submodules; this flattened module will then be compiled into the Maude rewrite engine. By systematically using the metaprogramming capabilities of Maude, we can both evaluate module expressions into structured module hierarchies and flatten such hierarchies into unstructured modules for execution. All such module operations are defined by equations that operate on the metalevel term representations of modules. This is essentially the idea behind the implementation of Full Maude in Maude.

In Full Maude, the use of module expressions is somewhat more general than in Core Maude. A Full Maude module expression can be used in any place where a module name is expected. Thus, as in Core Maude, in Full Maude, module expressions can be used as:

- arguments of a `protecting`, `extending`, or `including` importation,
- the source or target of a view, or
- the parameter of a module, provided the top level is a theory.

Furthermore, in Full Maude, they can also be used, e.g., to express the module in which a `red` or `rew` command will be executed,

```
Maude> (red in BOOL * (op true to T, op false to F) : T or F .)
result Bool :
  T
```

or as argument of any other command requiring a module name,

```
Maude> (show ops LIST{Nat} .)
  op $reverse : List{Nat}List{Nat}-> List{Nat}.
  op $size : List{Nat}Nat -> Nat .
  op append : List{Nat}List{Nat}-> List{Nat}.
  op append : List{Nat}NeList{Nat}-> NeList{Nat}.
  op append : NeList{Nat}List{Nat}-> NeList{Nat}.
  ...
```

Of course, this works with any module, and not only with predefined modules. For example, let us do the same with the instantiation of the `SET-MAX` module presented in Section 6.3.4 (which we assume is in file `set-max.maude`) with the view `IntAsToset` described in Section 6.3.2. Although we can use Core Maude modules in Full Maude, we do not have access to user-defined Core Maude views from Full Maude. Any such view must be entered into Full Maude before it is used in a module instantiation. Note that although Core Maude modules are implicitly entered into Full Maude's database, they are recompiled, and therefore, any view required for recompiling the corresponding module must also be entered. The evaluation of the module expression `SET-MAX{IntAsToset}` requires views `TOSET` and `IntAsToset`.

```
Maude> load set-max.maude
Maude> select FULL-MAUDE .
Maude> loop init .

                Full Maude 3.1 Oct 12 2020

Maude> (view TOSET from TRIV to TOSET is
         sort Elt to Elt .
       endv)
Introduced view TOSET

Maude> (view IntAsToset from TOSET to INT is
         sort Elt to Int .
       endv)
Introduced view IntAsToset

Maude> (red in SET-MAX{IntAsToset} : max((5, 4, 8, 4, 6, 5)) .)
result NzNat :
  8
```

Similarly, after entering the Full Maude version of the `RingToRat` view, we can reduce the same expression we reduced in Section 6.3.4 as follows:

```
Maude> (red in RAT-POLY{Qid} :
         (((2 / 3) (('X ^ 2) ('Y ^ 3)))
          ++ ((7 / 5) (('Y ^ 2) ('Z ^ 5))))
         (((1 / 7) ('U ^ 2)) ++ (1 / 2)) .)
result Poly{RingToRat,Qid} :
  (1/3('X ^ 2)'Y ^ 3)
  ++ (1/5('U ^ 2)('Y ^ 2)'Z ^ 5)
  ++ (2/21('U ^ 2)('X ^ 2)'Y ^ 3)
  ++ (7/10('Y ^ 2)'Z ^ 5)
```

As we will see below, a module expression can also be used as the parameter of a view, provided the top level is a theory.

21.3.1 The tuple and power module expressions

The evaluation of an n-tuple module expression consists in the generation of a parameterized functional module with the number of `TRIV` parameters specified by the argument n. A sort for tuples of such size, and the corresponding constructor `(_,...,_)` and selector operators `p1_, ..., pn_`, are also defined. For example, the module expression `TUPLE[2]` automatically generates as result the following module (notice the backquotes in the declaration of the tuple constructor).

```
(fmod TUPLE[2]{C1 :: TRIV, C2 :: TRIV} is
  sorts Tuple{C1, C2} .
  op `(_`,_`) : C1$Elt C2$Elt -> Tuple{C1, C2} [ctor].
  op p1_ : Tuple{C1, C2} -> C1$Elt .
  op p2_ : Tuple{C1, C2} -> C2$Elt .
  var E1 : C1$Elt .
  var E2 : C2$Elt .
  eq p1(E1, E2) = E1 .
  eq p2(E1, E2) = E2 .
endfm)
```

In the Clear [21] and OBJ [79] family of languages, module operations take theories, modules, and views, and return new theories and modules (see Chapter 6); on the other hand, the `TUPLE[_]` operation takes a nonzero natural number n and returns a parameterized `TUPLE[`n`]` module; this is impossible to achieve with the Clear/OBJ repertoire of module operations. Even though an n-tuple module expression is in principle of a completely different nature from the usual Clear/OBJ module operations, the way Full Maude handles it is the same as the way it handles any other module expression. Its evaluation produces a new unit, a parameterized functional module in this case, with the module expression as its name.

Suppose that we want to specify a library in which we have the information on the books in a record structure with the title, author, year of publication, publisher, and number of copies available. We may use a specification beginning as follows:

```
(fmod LIBRARY is
  pr TUPLE[5]{String, String, Nat, String, Nat}
       * (op p1_ to title,
          op p2_ to author,
          op p3_ to year,
          op p4_ to publisher,
          op p5_ to copies) .
  ---- ...
endfm)
```

The particular case of a tuple in which all component sorts are equal is provided by the n-power module expression. For example, the module expression `POWER[5]` automatically generates as result the following module:

```
(fmod POWER[5]{X :: TRIV} is
  protecting TUPLE[5]{X, X, X, X, X}
    * (sort Tuple{X, X, X, X, X} to Power{X}) .
endfm)
```

We can use the power module expression in any place where a module name is expected, like in a reduction

```
Maude> (red in POWER[10]{Nat} : p5 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) .)
result NzNat :
  4
```

or in an importation to build other modules:

```
(fmod PERSON-RECORD is
  pr POWER[3]{String}
       * (sort Tuple{String, String, String} to PersonRecord,
          op p1_ to firstname,
          op p2_ to lastname,
          op p3_ to address) .
  op fullName : PersonRecord -> String .
  vars F L A : String .
  eq fullName((F, L, A)) = F + " " + L .
endfm)

Maude> (red fullName(("John", "Smith", "Maude Ave")) .)
result String :
  "John Smith"
```

21.4 Moving up and down between reflection levels

The functions provided by Core Maude for moving up and down reflection levels (see Section 17.6.1) are not very useful in Full Maude. Although they are available as part of the module `META-LEVEL`, they take as one of their arguments the name of a module entered into Core Maude. Since the databases of modules are different, these functions work in Full Maude only for Core Maude predefined modules. Full Maude provides its own functions `upTerm` and `upModule` for moving, respectively, terms and modules up reflection levels, and an additional `down` command which allows moving terms down reflection levels.

Let us consider the following module for the examples in the coming sections.

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

In what follows we will use the notation $\overline{t}$ and $\overline{M}$ to refer to the metarepresentations of a term t and a module M, respectively. For example, we will write the metarepresentation of `0 + s 0` as $\overline{\texttt{0 + s 0}}$ instead of

```
'_+_['0.Nat, 's_['0.Nat]]
```

21.4.1 Up

As in Core Maude, in Full Maude we can use the `upModule` and `upTerm` functions to avoid the cumbersome task of explicitly writing, respectively, the metarepresentation of a module or the metarepresentation of a term in a given module. The Full Maude `upModule` function takes as a single argument the name of a module and returns its metarepresentation;[2] `upTerm` takes two arguments, the name of a module and a term in such a module, and returns the corresponding metarepresentation of the term.

Therefore, by evaluating in any module importing the module `META-LEVEL` the `upModule` function with the name of any module in the system—either in Core Maude or in Full Maude—as argument, we obtain the metarepresentation of such a module. For example, assuming that the previous module `NAT-PLUS` has been entered into Full Maude, and therefore it is in its database, we can get its metarepresentation, which we denoted by $\overline{\texttt{NAT-PLUS}}$, as follows:

```
Maude> (red in META-LEVEL : upModule(NAT-PLUS) .)
result FModule :
  fmod 'NAT-PLUS is
  nil
  sorts 'Bool ; 'Nat .
  none
  op '0 : nil -> 'Nat [none] .
  op '_+_ : 'Nat 'Nat -> 'Nat [assoc comm id('0.Nat)] .
  ...
```

We can use the metarepresentation obtained in this way in any place where a term of sort `Module` is expected. For example, we can apply the function `getOps` in `META-LEVEL` (see Section 17.4) to `upModule(NAT-PLUS)` as follows:

[2] The Core Maude `upModule` function takes as second argument a Boolean value (see Section 17.6.1).

```
Maude> (red in META-LEVEL : getOps(upModule(NAT-PLUS)) .)
result OpDeclSet :
  op '0 : nil -> 'Nat [none] .
  op '_+_ : 'Nat 'Nat -> 'Nat [assoc comm id('0.Nat)] .
  op '_=/=_ : 'Universal 'Universal -> 'Bool
       [poly(1 2)
        prec(51)
        special(
          id-hook('EqualitySymbol,nil)
          term-hook('equalTerm,'false.Bool)
          term-hook('notEqualTerm,'true.Bool))] .
  ...
```

Similarly, we can use it with descent functions as discussed in Section 17.6.

```
Maude> (red in META-LEVEL :
          metaReduce(upModule(NAT-PLUS), '_+_['0.Nat, 's_['0.Nat]]) .)
result ResultPair :
  {'s_['0.Nat],'Nat}
```

But, instead of explicitly writing the metarepresentation $\overline{\texttt{0 + s 0}}$ in the above reduction we can also make good use of the `upTerm` function, that allows us to get the metarepresentation of a term in a given module.

```
Maude> (red in META-LEVEL :
          metaReduce(upModule(NAT-PLUS), upTerm(NAT-PLUS, 0 + s 0)) .)
result ResultPair :
  {'s_['0.Nat], 'Nat}
```

As another example, to obtain the metarepresentation of the term `s 0` in the module `NAT-PLUS` above, which we denoted by $\overline{\texttt{s 0}}$, we can write

```
Maude> (red in META-LEVEL : upTerm(NAT-PLUS, s 0) .)
result GroundTerm :
  's_['0.Nat]
```

The module name is the first argument of the `upTerm` 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 to obtain the correct metarepresentation. Note also that the above reduction only makes sense at the metalevel, that is, either in the `META-LEVEL` module itself or in a module importing it.

21.4.2 Down

The result of a metalevel computation, that may use several levels of reflection, can be a term or a 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 `upTerm`, since it gives us back the original term from its metarepresentation. Notice that `down` is not a function, but a command instead, because it is more general, taking other commands as arguments, as we are going to explain.

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. The syntax of the `down` command is as follows:

`down` ⟨*ModuleExpression*⟩ `:` ⟨*Command*⟩

Thus, we can give the following command.

```
Maude> (down NAT-PLUS :
         red in META-LEVEL :
           getTerm(
             metaReduce(upModule(NAT-PLUS),
                        upTerm(NAT-PLUS, 0 + s 0))) .)
result Nat :
  s 0
```

Notice that this is equivalent to what we may write using the overline notation as:

```
Maude> red getTerm(metaReduce(\allttoverline{\texttt{NAT-PLUS}}, \allttoverline{\texttt{0 + s 0}})) .
result Term: \allttoverline{\texttt{s 0}}
```

The use of `upTerm` and `down` can be iterated with as many levels of reflection as we wish. For example, we can give the command

```
Maude> (red in META-LEVEL :
           getTerm(
             metaReduce(upModule(META-LEVEL),
               upTerm(META-LEVEL,
                 getTerm(
                   metaReduce(upModule(NAT-PLUS),
                              upTerm(NAT-PLUS, 0 + s 0)))))) .)
result GroundTerm :
  '_`[_`][''s_.Sort, ''0.Nat.Constant]
```

This is equivalent to what we would have written using the overline notation as

```
Maude> red getTerm(metaReduce(\allttoverline{\texttt{META-LEVEL}},
                   \allttoverline{\texttt{metaReduce(}\allttoverline{\texttt{NAT-PLUS}}\texttt{, }\alltt
result Term: \allttoverline{\allttoverline{\texttt{s 0}}}
```

We can write expressions involving simultaneously `down`, `upModule`, and `upTerm`:

```
Maude> (down NAT-PLUS :
         down META-LEVEL :
           red in META-LEVEL :
               getTerm(
                 metaReduce(upModule(META-LEVEL),
                   upTerm(META-LEVEL,
                     getTerm(
                       metaReduce(upModule(NAT-PLUS),
                                  upTerm(NAT-PLUS, 0 + s 0)))))) .)
result Nat :
  s 0
```

The metalevel function `downTerm` can also be used, but it is a Core Maude function, and therefore can only be used on Core Maude modules.

```
Maude> (down NAT-PLUS :
         red in META-LEVEL :
           downTerm(
             getTerm(
               metaReduce(upModule(META-LEVEL),
                 upTerm(META-LEVEL,
```

```
                getTerm(
                  metaReduce(upModule(NAT-PLUS),
                             upTerm(NAT-PLUS, 0 + s 0))))),
            'T:Term) .)
result Nat :
  s 0
```

21.5 Ax-coherence completion

As pointed out in Section 14.3, the equations used for variant generation in an admissible equational theory must be confluent, terminating, sort-decreasing, and *explicitly* Ax-coherent. The confluence, termination, and sort-decreasingness of equational Maude specifications are the typical executability requirements for equational specifications, and can be checked using Maude's Church-Rosser Checker (CRC) [54, 56] and Termination Checker (MTT) [47].

Regarding the Ax-coherence requirement, the situation is different for different purposes. Ax-coherence is not currently required for rewrite theories written in Maude because specifications are implicitly completed for Ax-coherence within Maude (see Section 4.8). It is not required either for tools like the CRC or the Coherence Checker (ChC), or for narrowing (see Chapter 15) which also automatically complete the specifications provided. However, rewrite theories are assumed to be explicitly Ax-coherent for variant generation and variant-based unification (Section 14.8), the reason being that there is a different treatment of extension variables in rewriting and narrowing (see [137] for more details). Indeed, we require coherence rather than ground coherence, since the latter is weaker and sufficient for rewriting but insufficient for narrowing or critical pair analysis.

For theories Ax that are combinations of associativity, commutativity, and identity axioms, we can make any specification explicitly Ax-coherent by using a procedure which adds Ax-extensions and always terminates (see [123], and Section 4.8 for a more informal explanation). The procedure followed to automatically complete for Ax-coherence for rewriting and for narrowing, or in the CRC or ChC tools is carried on differently. Whilst for rewriting the completion is performed by (Core) Maude, in the other cases the completion is performed by Maude code provided by Full Maude.

Full Maude provides the following `axCohComplete` operation in its module `AX-COHERENCE-COMPLETION`.

```
op axCohComplete : Module -> Module .
```

This operation takes (the metarepresentation of) a module which defines an order-sorted specification (i.e., no memberships are allowed) and returns (the metarepresentation of) another module whose equations and rules are modified to complete them modulo associativity and commutativity (AC), associativity, commutativity, and identity (ACU), associativity and identity (AU), associativity and left identity (AUl), associativity and right identity (AUr), and associativity (A).

More specifically, for each operator $f : S\ S \to S$ and equation/rule $f(t_1, \ldots, t_n) \to r$ *if* C in the module,

if f is AC	add $f(t_1,\ldots,t_n,x{:}[S]) \to f(r,x{:}[S])$ *if* C
if f is ACU	replace by $f(t_1,\ldots,t_n,x{:}[S]) \to f(r,x{:}[S])$ *if* C
if f is AU	replace by $f(x{:}[S],t_1,\ldots,t_n,y{:}[S]) \to f(x{:}[S],r,y{:}[S])$ *if* C
if f is AUl	replace by $f(x{:}[S],t_1,\ldots,t_n,y{:}[S]) \to f(x{:}[S],r,y{:}[S])$ *if* C
	add $f(x{:}[S],t_1,\ldots,t_n) \to f(x{:}[S],r)$ *if* C
if f is AUr	replace by $f(x{:}[S],t_1,\ldots,t_n,y{:}[S]) \to f(x{:}[S],r,y{:}[S])$ *if* C
	add $f(t_1,\ldots,t_n,y{:}[S]) \to f(r,y{:}[S])$ *if* C
if f is A	add $f(x{:}[S],t_1,\ldots,t_n,y{:}[S]]) \to f(x{:}[S],r,y{:}[S])$ *if* C
	add $f(x{:}[S],t_1,\ldots,t_n) \to f(x{:}[S],r)$ *if* C
	add $f(t_1,\ldots,t_n,y{:}[S]) \to f(r,y{:}[S])$ *if* C

To deal with collapses, those rules/equations of the form $l \to r$ *if* C that do not match any of the above cases, and such that there is an operator $f : S\ S \to S$ with identity and with S and $leastSort(l)$ in the same kind, are completed following the above cases as if it were the equation/rule $f(l) \to r$ *if* C.

Ax-completion is available in Full Maude through the following command:

```
(ax coherence completion [ ⟨ModuleExpression⟩ ] .)
```

where ⟨*ModuleExpression*⟩ is any module expression. As usual, if no module expression is given the default current module is completed.

For example, let us consider the following non-coherent version of the equational theory for exclusive or (see Section 14.1).

```
mod EXCLUSIVE-OR-NOT-COHERENT is
  sorts Nat NatSet .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  subsort Nat < NatSet .
  op mt : -> NatSet .
  op _*_ : NatSet NatSet -> NatSet [ctor assoc comm] .

  vars X Y Z : [NatSet] .
  eq [idem] : X * X = mt [variant] .
  eq [id] : X * mt = X [variant] .
endm
```

The module `EXCLUSIVE-OR-NOT-COHERENT` is not AC-coherent. The command

```
Maude> (ax coherence completion EXCLUSIVE-OR-NOT-COHERENT .)
```

shows the completed version of the `EXCLUSIVE-OR-NOT-COHERENT` module. Specifically, the following equation is *added*:

```
eq [idem] : X:[NatSet] * X:[NatSet] * X@@@:[NatSet]
  = mt * X@@@:[NatSet] [variant] .
```

Notice the new variable `X@@@:[NatSet]`. The resulting module is equivalent to the module `EXCLUSIVE-OR` in Section 14.1.

As another example, the Ax-completion of the `VENDING-MACHINE` module in Section 5.1 results in a module with the same definitions but with the `change` rule *replaced* by the rule

```
rl [change] : q q q q X@@@:[Marking] => $ X@@@:[Marking] .
```

In general, of course, a module can have operators with any combination of associativity, commutativity, and identity, with associated equations and rules. In that case the `ax`

`coherence completion` command will add or replace the corresponding completion equations and rules.

The narrowing command (see Section 15.6) and the CRC and ChC tools use the metalevel operation `axCohComplete` to complete the modules before using them. As any other function in Full Maude, it is available for its direct use in other tools or commands. We can illustrate the use of this metalevel function with the coherence completion of the `EXCLUSIVE-OR-NOT-COHERENT` module:

```
Maude> reduce in FULL-MAUDE :
        axCohComplete(upModule('EXCLUSIVE-OR-NOT-COHERENT, false)) .

result SModule :
  mod 'EXCLUSIVE-OR-NOT-COHERENT is
  including 'BOOL .
  sorts 'Nat ; 'NatSet .
  subsort 'Nat < 'NatSet .
  op '0 : nil -> 'Nat [ctor] .
  op 'mt : nil -> 'NatSet [ctor] .
  op '_*_ : 'NatSet 'NatSet -> 'NatSet [assoc comm ctor] .
  op 's : 'Nat -> 'Nat [ctor] .
  none
  eq '_*_['X:`[NatSet`],'mt.NatSet]
    = 'X:`[NatSet`] [label('id) variant] .
  eq '_*_['X:`[NatSet`],'X:`[NatSet`]]
    = 'mt.NatSet [label('idem) variant] .
  eq '_*_['X:`[NatSet`],'X:`[NatSet`],'X@@@:`[NatSet`]]
    = '_*_['mt.NatSet,'X@@@:`[NatSet`]] [label('idem) variant] .
  none
endm
```

The following example shows the completion of rules that require changing the top operator of the left-hand side term.

```
mod NARROWING-VM-NOTOP is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  subsort Marking < State .
  ops $ q : -> Coin . ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  eq [change]: q q q q = $ [variant] .
endm

Maude> reduce in FULL-MAUDE :
        axCohComplete(upModule('NARROWING-VM-NOTOP, false)) .

result SModule:
  mod 'NARROWING-VM-NOTOP is
    including 'BOOL .
    sorts 'Coin ; 'Item ; 'Marking ; 'Money ; 'State .
```

```
  subsort 'Coin < 'Money .
  subsort 'Item < 'Marking .
  subsort 'Marking < 'State .
  subsort 'Money < 'Marking .
  op '$ : nil -> 'Coin [none] .
  op '__ : 'Marking 'Marking -> 'Marking [assoc comm id('empty.Money)] .
  op '__ : 'Money 'Money -> 'Money [assoc comm id('empty.Money)] .
  op 'a : nil -> 'Item [none] .
  op 'c : nil -> 'Item [none] .
  op 'empty : nil -> 'Money [none] .
  op 'q : nil -> 'Coin [none] .
  none
  eq '__['q.Coin,'q.Coin,'q.Coin,'q.Coin,'X@@@:`[State`]]
    = '__['$.Coin,'X@@@:`[State`]]
    [variant label('change)] .
  rl '__['$.Coin,'X@@@:`[State`]]
    => '__['c.Item,'X@@@:`[State`]]
    [label('buy-c)] .
  rl '__['$.Coin,'X@@@:`[State`]]
    => '__['a.Item,'q.Coin,'X@@@:`[State`]]
    [label('buy-a)] .
endm
```

No variant unifier

21.6 Differences between Full Maude and Core Maude

Apart from those features available in Full Maude that are not supported in Core Maude (discussed above in Sections 21.3-21.5 and later in Chapters 15 and 22), we find a number of differences between Full Maude and Core Maude. There are some obvious ones, like the fact that any input enclosed in parentheses is handled by Full Maude. Core Maude gives a much more precise information on error location and cause, and do not have the overload of Full Maude when tracing and debugging. Thus, Full Maude modules, theories, views, and commands can refer to modules, theories, and views in both Core Maude and Full Maude's databases. There are also differences in pretty-printing, tracing, debugging, etc. Moreover, there are also other differences that impose certain limitations on the specifications themselves:

1. External objects are not accessible in Full Maude.

2. Operator and message names have to be given in their equivalent *single-identifier form* when they are declared (see below), but they can later be written in the usual way in statements and in terms for evaluation.

3. Sort names used in term qualifications, membership assertions, and on-the-fly declarations of variables have to be in their equivalent *single-identifier form*.

4. The `continue`, `show component`, `show path`, and `show search graph` commands are not supported in Full Maude.

5. Full Maude does not support external objects.

In the rest of the section we explain the first two restrictions in some detail and give some hints on how to avoid them.

An operator name has to be given as a single identifier; multi-identifier operators have to be declared in their single-identifier form, that is, each identifier in a multi-identifier name has to be preceded by a backquote. For example, to define an operator with name `_less than or equal_`, we have to declare it in its single identifier form `_less‘than‘or‘equal_`. Except for having to use the single-identifier form in the operator name, the declaration of operators is exactly as in Core Maude. For example, the declaration of this operator on sort, say, `Int` is as follows.

```
op _less‘than‘or‘equal_ : Int Int -> Bool .
```

Remember that not only blank spaces, but also the special characters ‘`{`’, ‘`}`’, ‘`(`’, ‘`)`’, ‘`[`’, ‘`]`’ and ‘`,`’ break the identifiers. Therefore, to declare in Full Maude an operator such as `{_}` taking an element of sort, say, `Int` and with value sort `Set`, we should add appropriate backquotes, as follows:

```
op ‘{_‘} : Int -> Set .
```

As in Core Maude, several operators with the same arity and coarity can be defined in the same declaration using the keyword `ops`, but, again, each operator name has to be given in its single-identifier form. We could have for example the following declaration.

```
ops _‘{_‘} _‘,_ : Foo Bar -> Baz .
```

Since each operator name is a single identifier, parentheses are not needed to indicate the boundaries between the syntactic forms of the different operators.

As for operator names, message names can be mixfix, but they have to be declared in single-identifier form. Thus, to define a message `credit` in an object-oriented module (see Chapter 22) with syntax, say, `(_)credit_` the declaration has to be given as follows.

```
msg ‘(_‘)credit_ : Oid Nat -> Msg .
```

And the same applies to declarations of multiple message names:

```
msgs ‘(_‘)credit_ ‘(_‘)debit_ : Oid Nat -> Msg .
```

The second problem mentioned at the beginning of this section affects the qualification of terms by sort names, on-the-fly declarations of variables, and membership assertions. In these three situations, the user must use the names of parameterized sorts, not as he or she has defined them, but in their equivalent single-identifier form. Thus, if we have, for example, a sort `List{Nat}` and a constant `nil` in it, if necessary, it should be qualified as `(nil).List‘{Nat‘}`. A variable `L` of sort `List{Nat}` being declared on the fly should be written `L:List‘{Nat‘}`. Similarly, to check whether a term `T` has sort `List{Nat}` we have to write `T : List‘{Nat‘}` or `T :: List‘{Nat‘}`, depending on the kind of sort check we wish to perform.

Chapter 22

Object-Oriented Modules

In Full Maude, concurrent object-oriented systems can be defined by means of *object-oriented modules*—introduced by the keyword `omod...endom`—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.

As in Core Maude, we may have specifications of object-oriented systems in system modules; for example, we could enter into Full Maude the system modules describing object-based systems discussed in Chapter 8 by enclosing them in parentheses. However, although Maude's system modules are sufficient for specifying object-oriented systems, there are important conceptual advantages provided by Full Maude's syntax for object-oriented modules. Such syntax allows the user to think and express his/her thoughts in object-oriented terms whenever such a viewpoint seems best suited for the problem at hand. Those conceptual advantages would be partially lost if only system modules at the Core Maude level were provided.

Object-oriented modules are however just syntactic sugar: they are internally transformed into system modules for execution purposes (Section 22.8). All object-oriented modules implicitly include the `CONFIGURATION` module (see Section 8.1), and thus assume the latter's syntax. Recall that the module `CONFIGURATION` defines the basic concepts of concurrent object systems; among others, and besides the `Configuration` sort, it includes the declarations of sorts

- `Oid` of object identifiers,
- `Cid` of class identifiers,
- `Object` for objects, and
- `Msg` for messages.

22.1 Object-oriented systems

Some object-oriented concepts were introduced in Chapter 8. Here we recall some of them and then focus on the notions of class and inheritance, and on the additional syntactic facilities provided by Full Maude to support object-oriented programming.

22.1.1 Objects and messages

As in Core Maude, an *object* in a given state is represented as a term of the form

```
< O : C | ⟨att-1⟩, ... , ⟨att-n⟩ >
```

but Full Maude supports and enforces a specific choice for the syntax of attributes. Each attribute of sort `Attribute` consists of a *name* (attribute identifier), followed by a colon '`:`' (with white space both before and after), followed by its *value*, which must have a given sort. Therefore, the Full Maude syntax for objects is

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

where `O` is the object's name or identifier, `C` is its class identifier, the `ai`'s are the names of the object's *attribute identifiers*, and the `vi`'s are the corresponding *values*, for $i = 1 \ldots n$. In particular, an object with no attributes can be represented as

```
< O : C | >
```

Messages do not have a fixed syntactic form; their syntax is instead defined by the user for each application. The only assumption made by the system is that the first argument of a message is the identifier of its destination object. Messages satisfying this requirement should be declared using the `msg` keyword. It is still possible to declare messages not following this requirement as operators of sort `Msg`; but, if declared as operators, no `message` attribute will be provided for them (see Sections 8.1 and 8.2). For example, the following declarations of messages are possible.

```
msg credit : Oid Nat -> Msg .
op left : -> Msg .
```

The concurrent state of an object-oriented system is then a multiset of objects and messages, of sort `Configuration`, with multiset union described with empty syntax `__`, and with `assoc`, `comm`, and `id: none` as operator attributes.

22.1.2 Classes

Classes are defined with the keyword `class`, followed by the name of the class, followed by a bar '`|`', followed by a list of attribute declarations separated by commas. Each attribute declaration has the form $a : S$, where a is an attribute identifier and S is the sort in which the values of the attribute identifier range. That is, class declarations have the form

```
class C | a1 : ⟨Sort-1⟩, ... , an : ⟨Sort-n⟩ .
```

In particular, we can declare classes without attributes using syntax

```
class C .
```

Class names have the same form as sort names. In particular, class names may be parameterized in a way completely similar to parameterized sort names (see Sections 6.3.3, 22.3, and 22.4).

As an example of class declaration, the class `Account` of bank account objects with a `bal`ance attribute, introduced in Section 8.1, is now declared as follows:

```
class Account | bal : Int .
```

As another example, a class `Person`, with a name, an age, and a bank account as attributes can then be declared as follows:

```
class Person | name : String, age : Nat, account : Oid .
```

In this case, a person has a reference to his/her account in an `account` attribute of sort `Oid`.

All Full Maude object-oriented modules have an operation `class` that takes an object as argument and returns its actual class. Thus, for example,

```
getClass(< A-002 : Account | bal : 1000 >)
```

returns the class identifier `Account`. This operation will be particularly useful when combined with the inheritance relation (see the use of the `getClass` operation in the example in Section 22.5).

The syntax for message declarations is similar to the syntax for the declaration of operators, but using `msg` and `msgs` instead of `op` and `ops`, and having as result sort `Msg` or a subsort of it. Thus, `msg` is used to declare a single message, and `msgs` may be used for declaring multiple messages. The user can introduce subsorts of the predefined sort `Msg`, so that it is possible to declare messages of different types. This may be useful for restricting the kind of messages that could be received by a particular type of object. As in the case of operators, messages can be overloaded and can be declared with operator attributes.

In the account example, the three kinds of messages involving accounts are `credit`, `debit`, and `from_to_transfer_`, whose user-definable syntax is introduced in the following declarations:

```
msgs credit debit : Oid Nat -> Msg .
msg from_to_transfer_ : Oid Oid Nat -> Msg .
```

Notice the use of the `Oid` sort for specifying the addressee of a message, as in `credit` and `debit`, and for specifying the objects involved in a message, e.g., the source and the target accounts of an account transfer in the `from_to_transfer_` message. Note that, as explained in Chapter 8 for Core Maude, Full Maude also assumes that the message's destination object is the first argument mentioned in the message declaration. This convention is needed by the object-message fair rewriting strategy (see Section 8.2). The behavior associated with the messages is then specified by rewrite rules in a declarative way (see Section 22.1.4).

Given object identifiers `Smith` and `A-002`, the following term may represent a configuration with a person, his account, and a `credit` message sent to it.

```
< Smith : Person | name : "John", age : 34, account : A-002 >
< A-002 : Account | bal : 1000 >
credit(A-002, 100)
```

22.1.3 Inheritance

Class inheritance is directly supported by Maude's order-sorted type structure. Since class names are a particular case of sort names, a subclass declaration `C < C'` in an object-oriented module is just 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, together with the newly defined attributes, messages, and rules of the subclass, characterize the structure and behavior of the objects in the subclass.

For example, we can define an object-oriented module `SAVING-ACCOUNT` of saving accounts introducing a subclass `SavingAccount` of `Account` with a new attribute `rate` recording the interest rate of the account.

```
class SavingAccount | rate : Float .
subclass SavingAccount < Account .
```

In this example there is only one class immediately above `SavingAccount`, namely, `Account`. In general, however, a class C may be defined as a subclass of several classes $D_1, \ldots, D_k$, i.e., *multiple inheritance* is supported. If an attribute and its sort have already been declared in a superclass, they should not be declared again in the subclass. Indeed, all such attributes are *inherited*. In the case of multiple inheritance, the only requirement that is made is that

if an attribute occurs in two different superclasses, then the sort associated with it in each of those superclasses must be the same.[1] In summary, a class inherits all the attributes, messages, and rules from all its superclasses. An object in the subclass behaves exactly as any object in any of the superclasses, but it may exhibit additional behavior due to the introduction of new attributes, messages, and rules in the subclass.

Objects in the class `SavingAccount` will have an attribute `bal` and can receive messages debiting, crediting and transferring funds exactly as any other object in the class `Account`. For example, the following object is a valid instance of class `SavingAccount`.

```
< A-002 : SavingAccount | bal : 5000, rate : 3.0 >
```

As for subsort relationships, we can declare multiple subclass relationships in the same declaration. Thus, given classes `A`, ..., `H`, we can have a declaration of subclasses such as

```
subclasses A B C < D E < F G H .
```

22.1.4 Object-oriented rules

The behavior associated with messages is specified by rewrite rules in a declarative way. For example, the semantics of the `credit`, `debit`, and `from_to_transfer_` messages declared in Section 22.1.2 may be given as follows:

```
vars A B : Oid .
var  M : Nat .
vars N N' : Int .

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

Note that the multiset structure of the configuration provides the top-level distributed structure of the system and allows concurrent application of the rules. The reader is referred to [104] for a detailed explanation of the logical semantics of the object-oriented model of computation supported by Maude.

In object-oriented modules it is possible not to mention in a given rule those attributes of an object that are not relevant for that rule. The attributes mentioned only on the lefthand side of a rule are preserved unchanged, the original values of attributes mentioned only on the

[1] If a class inherits from two different superclasses that share an attribute but with different associated sorts, then both attributes are inherited in the subclass, thus muddling them up.

righthand side do not matter, and all attributes not explicitly mentioned are left unchanged (see Section 22.8 for more details). For instance, a message for changing the age of a person defined by the class `Person` (introduced in Section 22.1.2) may be defined as follows:

```
msg to_:‘new‘age_ : Oid Nat -> Msg .
var A : Nat .
var O : Oid .

rl [change-age] :
  < O : Person | >
  to O : new age A
  => < O : Person | age : A > .
```

The attributes `name` and `account`, which are not mentioned in this rule, are not changed when applying the rule. The value of the `age` attribute is replaced by the given new age, independently of its previous value.

The following module `ACCOUNT` contains all the declarations above defining the class `Account`. Note that `Qid` is declared as a subsort of `Oid`, making any quoted identifier a valid object identifier.

```
(omod ACCOUNT is
  protecting QID .
  protecting INT .

  subsort Qid < Oid .
  class Account | bal : Int .
  msgs credit debit : Oid Int -> Msg .
  msg from_to_transfer_ : Oid Oid Int -> Msg .

  vars A B : Oid .
  var  M : Nat .
  vars N N' : Int .

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

We can now rewrite a simple configuration consisting of an account and a message as follows:

```
Maude> (rew < 'A-06238 : Account | bal : 2000 >
```

```
           debit('A-06238, 1000) .)

result Object :
  < 'A-06238 : Account | bal : 1000 >
```

The following module contains the declarations for the class `SavingAccount`.

```
(omod SAVING-ACCOUNT is
  including ACCOUNT .
  protecting FLOAT .
  class SavingAccount | rate : Float .
  subclass SavingAccount < Account .
endom)
```

We leave unspecified the rules for computing and crediting the interest of an account according to its rate, whose proper expression should introduce a real-time[2] attribute in account objects.

We can now rewrite a configuration, obtaining the following result.

```
Maude> (rew < 'A-73728 : SavingAccount | bal : 5000, rate : 3.0 >
           < 'A-06238 : Account | bal : 2000 >
           < 'A-28381 : SavingAccount | bal : 9000, rate : 3.0 >
           debit('A-06238, 1000)
           credit('A-73728, 1300)
           credit('A-28381, 200) .)

result Configuration :
           < 'A-06238 : Account | bal : 1000 >
           < 'A-73728 : SavingAccount | bal : 6300, rate : 3.0 >
           < 'A-28381 : SavingAccount | bal : 9200, rate : 3.0 >
```

We can also search over configurations. In this case the search pattern takes into account object-oriented information, finding also states where a subclass matches the pattern. For example, we can look for final states having accounts with balance less than 8000 with the following command:

```
Maude> (search in SAVING-ACCOUNT :
           < 'A-73728 : SavingAccount | bal : 5000, rate : 3.0 >
           < 'A-06238 : Account | bal : 2000 >
           < 'A-28381 : SavingAccount | bal : 9000, rate : 3.0 >
           debit('A-06238, 1000)
           credit('A-73728, 1300)
           credit('A-28381, 200)
        =>! C:Configuration
            < O:Oid : Account | bal : N:Nat >
        such that N:Nat < 8000 .)
search in SAVING-ACCOUNT :
  < 'A-73728 : SavingAccount | bal : 5000, rate : 3.0 >
  < 'A-06238 : Account | bal : 2000 >
  < 'A-28381 : SavingAccount | bal : 9000, rate : 3.0 >
  debit('A-06238, 1000)
  credit('A-73728, 1300)
  credit('A-28381, 200)
  =>! C:Configuration
```

[2]See [120, 122] for a general methodology to specify real-time systems, including object-oriented ones, in rewriting logic.

```
      < O:Oid : V#0:Account | bal : N:Nat, V#1:AttributeSet > .

Solution 1
C:Configuration -->
  < 'A-28381 : SavingAccount | bal : 9200,rate : 3.0 >
  < 'A-73728 : SavingAccount | bal : 6300,rate : 3.0 > ;
N:Nat --> 1000 ;
O:Oid --> 'A-06238 ;
V#0:Account --> Account ;
V#1:AttributeSet --> (none).AttributeSet

Solution 2
C:Configuration -->
  < 'A-06238 : Account | bal : 1000 >
  < 'A-28381 : SavingAccount | bal : 9200, rate : 3.0 > ;
N:Nat --> 6300 ;
O:Oid --> 'A-73728 ;
V#0:Account --> SavingAccount ;
V#1:AttributeSet --> rate : 3.0

No more solutions.
```

Notice that the search pattern has been transformed so that objects in subclasses match. In this example, we obtain as solutions both an object of class `Account` and an object in the subclass `SavingAccount`.

22.2 Example: a rent-a-car store

In order to further illustrate Full Maude's object-oriented features, we specify a simple rent-a-car store example. Several rules in this specification have variables in their righthand sides or conditions not present in their lefthand sides; therefore, these rules are not directly executable by the rewrite engine and are declared as nonexecutable. In order to run the object-oriented system, we will have to use strategies; a possible such strategy will be presented in Section 22.6.

The regulations of the system, especially those that govern the rental processes, are the following:

1. Cars are rented for a specific number of days, after which they should be returned.
2. A car can be rented only if it is available.
3. No credit is allowed; customers must pay cash.
4. Customers must make a deposit at pick-up time of the estimated rental charges.
5. Rental charges depend on the car class. There are three categories: economy, mid-size, and full-size cars.
6. When a rented car is returned, the deposit is used to pay the rental charges.
7. If a car is returned before the due date, the customer is charged only for the number of days the car has been used. The rest of the deposit is reimbursed to the customer.
8. Customers who return a rented car after its due date are charged for all the days the car has been used, with an additional 20% charge for each day after the due date.

9. Failure to return the car on time or to pay a debt may result in the suspension of renting privileges.

Let us begin with the static aspects of this system, i.e., with its structure. We can identify three main classes, namely the store, customer, and car classes. There are three kinds of cars: economy, mid-size, and full-size cars.

Customers may rent cars. This relationship may be represented by a `Rental` class which, in addition to references to the objects involved in the relationship, has some extra attributes. The system also requires some control over time, which we get with a class representing calendars that provides the current date and simulates the passage of time.

The `Customer` class has three attributes, namely, `suspended`, `cash`, and `debt` to keep track of, respectively, whether he is suspended or not, the amount of cash that the customer currently has, and his debt with the store. Such a class is defined by the following Maude declaration:

```
class Customer | cash : Nat, debt : Nat, suspended : Bool .
```

The attribute `available` of the `Car` class indicates whether the car is currently available or not, and `rate` records the daily rental rate. We model the different types of cars for rent by three different subclasses, namely, `EconomyCar`, `MidSizeCar` and `FullSizeCar`.

```
class Car | available : Bool, rate : Nat .
class EconomyCar .
class MidSizeCar .
class FullSizeCar .
subclasses EconomyCar MidSizeCar FullSizeCar < Car .
```

Each object of class `Rental` will establish a relationship between a customer and a car, whose identifiers are kept in attributes `customer` and `car`, respectively. In addition to these, the class `Rental` is also declared with attributes `deposit`, `pickUpDate`, and `dueDate` to store, respectively, the amount of money left as a deposit by the customer, the date in which the car is picked up by the customer, and the date in which the car should be returned to the store.

```
class Rental | deposit : Nat, dueDate : Nat, pickUpDate : Nat,
               customer : Oid, car   : Oid .
```

Given the simple use that we are going to make of dates, we can represent them, for example, as natural numbers. Then, we may have a calendar object that keeps the current date and gets increased by a rewrite rule as follows:

```
class Calendar | date : Nat .
rl [new-day] :
  < O : Calendar | date : F >
  => < O : Calendar | date : F + 1 > .
```

We do not worry here about the frequency with which the date gets increased, the possible synchronization problems in a distributed setting, or with any other issues related to the specification of time. See the papers [120, 122] on the specification of real-time systems in rewriting logic and Maude for a discussion on these issues.

Four actions can be identified in our example:

- a customer rents a car,
- a customer returns a rented car,
- a customer is suspended for being late in paying her debt or for being late in returning a rented car, and

- a customer pays (part of) her debt.

The rental of a car by a customer is specified by the `car-rental` rule below, which involves the customer renting the car, the car itself (which must be available, i.e., not currently rented), and a calendar object supplying the current date. The rental can take place if the customer is not suspended, that is, if her identifier is not in the set of identifiers of suspended customers of the store, and if the customer has enough cash to make the corresponding deposit. Notice that a customer could rent a car for less time she really is going to use it on purpose, because either she does not have enough money for the deposit, or prefers making a smaller deposit. According to the description of the system, the payment takes place when returning the car, although there is an extra charge for the days the car was not reserved.

```
crl [car-rental] :
  < U : Customer | cash : M, suspended : false >
  < I : Car | available : true, rate : Rt >
  < C : Calendar | date : Today >
  => < U : Customer | cash : M - Amnt >
     < I : Car | available : false >
     < C : Calendar | >
     < A : Rental | pickUpDate : Today, dueDate : Today + NumDays,
         car : I, deposit : Amnt, customer : U, rate : Rt >
  if Amnt := Rt * NumDays /\ M >= Amnt
  [nonexec] .
```

Note that, as already mentioned, those attributes of an object that are not relevant for a rule do not need to be mentioned. Attributes not appearing in the righthand side of a rule will maintain their previous values unmodified. Furthermore, since the variables `A` and `NumDays` appear in the righthand side or condition of the rule but not in its lefthand side, this rule has to be declared as `nonexec`. Note as well the use of the attributes `customer` and `car` in objects of class `Rental`, which makes explicit that a rental relationship is between the customer and the car specified by these attributes.

A customer returning a car late cannot be forced to pay the total amount of money due at return time. Perhaps she does not have such an amount of money in hand. The return of a rented car is specified by the rules below. The first rule handles the case in which the car is returned on time, that is, the current date is smaller or equal to the due date, and therefore the deposit is greater or equal to the amount due.

```
crl [on-date-car-return] :
  < U : Customer | cash : M >
  < I : Car | rate : Rt >
  < A : Rental | customer : U, car : I, pickUpDate : PDt,
                 dueDate : DDt, deposit : Dpst >
  < C : Calendar | date : Today >
  => < U : Customer | cash : (M + Dpst) - Amnt >
     < I : Car | available : true >
     < C : Calendar | >
  if (Today <= DDt) /\ Amnt := Rt * (Today - PDt)
  [nonexec] .
```

In this case, part of the deposit needs to be reimbursed. We can see that the `Rental` object disappears in the righthand side of the rule, that is, it is removed from the set of rentals and the availability of the car is restored.

The second rule deals with the case in which the car is returned late.

```
crl [late-car-return] :
```

```
  < U : Customer | debt : M >
  < I : Car | rate : Rt >
  < A : Rental | customer : U, car : I, pickUpDate : PDt,
                 dueDate : DDt, deposit : Dpst >
  < C : Calendar | date : Today >
  => < U : Customer | debt : (M + Amnt) - Dpst >
     < I : Car | available : true >
     < C : Calendar | >
  if DDt < Today  *** it is returned late
     /\ Amnt := Rt * (DDt - PDt)
               + ((Rt * (Today - DDt)) * (100 + 20)) quo 100
  [nonexec] .
```

In this case, the customer's debt is increased by the portion of the amount due not covered by the deposit.

Debts may be paid at any time, the only condition being that the amount paid is between zero and the amount of money owed by the customer at that time.

```
crl [pay-debt] :
  < U : Customer | debt : M, cash : N >
  => < U : Customer | debt : M - Amnt, cash : N - Amnt >
  if 0 < Amnt /\ Amnt <= N /\ Amnt <= M
  [nonexec] .
```

Customers who are late in returning a rented car or in paying their debts "may" be suspended. The first rule deals with the case in which a customer has a pending debt, and the second one handles the case in which a customer is late in returning a rented car.

```
crl [suspend-late-payers] :
  < U : Customer | debt : M, suspended : false >
  => < U : Customer | suspended : true >
  if M > 0 .

crl [suspend-late-returns] :
  < U : Customer | suspended : false >
  < I : Car | >
  < A : Rental | customer : U, car : I, dueDate : DDt >
  < C : Calendar | date : Today >
  => < U : Customer | suspended : true >
     < I : Car | >
     < A : Rental | >
     < C : Calendar | >
  if DDt < Today .
```

Since the system is not terminating, and there are several rules with variables in their right-hand sides or conditions not present in their lefthand sides and not satisfying the admissibility conditions discussed in Section 5.3, strategies are necessary for controlling its execution. We can define many different strategies and use them in many different ways (see Section 17.7); a concrete possibility will be described later in Section 22.6.

22.3 Object-oriented parameterized programming

The notions of theory, view, and parameterized module (see Section 6.3) have been extended to the object-oriented case. In this section, we explain how to write object-oriented theories, views

with object-oriented theories as sources and object-oriented modules or object-oriented theories as targets, and object-oriented parameterized modules with possibly object-oriented theories as parameters. In Section 22.4 we explain how the module operations available in Full Maude have been extended, so that they are also available on object-oriented modules. In particular, we will see how it is possible to rename an object-oriented module and to instantiate an object-oriented module parameterized by an object-oriented theory with a view having another object-oriented module as its target.

22.3.1 Theories

In addition to functional and system theories, Full Maude also supports *object-oriented theories*. Their structure is the same as that of object-oriented modules. Object-oriented theories can have classes, subclass relationships, and messages. These object-oriented notions may be useful for the definition of theories; for example, the following theory `CELL` specifies the theory of classes with at least one attribute of any sort.

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

22.3.2 Views

For a view having an object-oriented theory as its source, the mapping of a class `C` in the source theory to a class `C'` in the target is expressed with syntax

```
class C to C' .
```

Attribute maps have the form

```
attr C . A to A' .
```

where `A` is the name of an attribute of class `C` in the source theory and `A'` is an attribute of the image class of `C` under the view.

The mapping of messages is expressed with syntax

```
msg M to M' .
```

where `M` is a message identifier or a message identifier together with its arity and value sort. As for operators, a message map in which explicit arity and coarity are given affects the entire family of subsort-overloaded message declarations associated with the declaration of the given message.

22.3.3 Parameterized object-oriented modules

Like any other type of module, object-oriented modules can be parameterized, and, like sort names, class names may also be parameterized. The naming of parameterized classes follows the same conventions discussed in Section 6.3.3 for parameterized sorts.

As an example of an object-oriented parameterized module, we define a stack of elements. We define a class `Stack{X}` as a linked sequence of node objects. Objects of class `Stack{X}` have a single attribute `first`, containing the identifier of the first node in the stack. If the stack is empty, the value of the `first` attribute is `null`. Each object of class `Node{X}` has an attribute `next` holding the identifier of the next node—which should be `null` if there is no next node—and an attribute `contents` to store a value of sort `X$Elt`. Notice that node identifiers

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 removal 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 OO-STACK{X :: TRIV} is
  protecting INT .
  protecting QID .
  subsort Qid < Oid .
  class Node{X} | next : Oid, contents : X$Elt .
  class Stack{X} | first : Oid .
  msg _push_ : Oid X$Elt -> Msg .
  msg _pop : Oid -> Msg .
  msg _top_ : Oid Oid -> Msg .
  msg _elt_ : Oid X$Elt -> Msg .

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

  vars O O' O'' : Oid .
  var  E : X$Elt .
  var  N : Int .

  rl [top] : *** top on a non-empty 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 non-empty 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 non-empty stack
     < O : Stack{X} | first : O' >
     < O' : Node{X} | next : O'' >
     (O pop)
     => < O : Stack{X} | first : O'' > .
endom)
```

Notice that `top` and `pop` messages are not received if the stack is empty.

We may want to define stacks storing not just 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 the parameterized module `OO-STACK2` below, which is parameterized by the object-oriented theory `CELL` introduced in Section 22.3.1. Notice that the main difference with respect to the previous `STACK` version is in the attribute `node`, that keeps the identifier of the object where the contents can be found instead of the attribute `contents` that provided direct access to those contents.

```
(omod OO-STACK2{X :: CELL} is
  protecting 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 X$Elt -> Msg .

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

  vars O O' O'' O''' : Oid .
  var  E : X$Elt .
  var  N : Int .

  rl [top] : *** top on a non-empty stack
     < O : Stack{X} | first : O' >
     < O' : Node{X} | node : O'' >
     < O'' : X$Cell | contents : E >
     (O top O''')
     => < O : Stack{X} | >
        < O' : Node{X} | >
        < O'' : X$Cell | >
        (O''' elt E) .

  rl [push1] : *** push on a non-empty 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 non-empty stack
     < O : Stack{X} | first : O' >
     < O' : Node{X} | next : O'' >
     (O pop)
     => < O : Stack{X} | first : O'' > .
```

```
endom)
```

22.4 Module operations on object-oriented modules

The module operations of summation, renaming, and instantiation have been extended so that they are also available on object-oriented modules.

22.4.1 Module summation and renaming

The summation and renaming of object-oriented modules is similar to their non-object-oriented counterparts. Renaming maps, however, are in this case available for mapping classes, attributes, and messages. Therefore, in addition to the renamings available in Core Maude, Full Maude also supports renaming maps of the form:

`class` ⟨*identifier*⟩ `to` ⟨*identifier*⟩
`attr` ⟨*class-identifier*⟩ `.` ⟨*attr-identifier*⟩ `to` ⟨*class-identifier*⟩
`msg` ⟨*identifier*⟩ `to` ⟨*identifier*⟩
`msg` ⟨*identifier*⟩ `:` ⟨*type-list*⟩ `->` ⟨*type*⟩ `to` ⟨*identifier*⟩

We illustrate the renaming of object-oriented modules with the following example:[3]

```
Maude> (show module OO-STACK2 * (class Stack{X} to Stack{X},
                                 class Node{X} to Node{X},
                                 attr Stack{X} . first to head,
                                 msg _elt_ to element,
                                 sort Int to Integer) .)

omod OO-STACK2 * (sort Int to Integer,
              msg _elt_ to element,
              class Node‘{X‘} to Node‘{X‘},
              class Stack‘{X‘} to Stack‘{X‘},
              attr Stack‘{X‘} . first to head) {X :: CELL} is
  protecting QID .
  protecting INT * (sort Int to Integer) .
  including CONFIGURATION+ .
  including CONFIGURATION .
  protecting BOOL .
  subsort Qid < Oid .
  class Node‘{X‘} | next : Oid, node : Oid .
  class Stack‘{X‘} | head : Oid .
  op null : -> Oid .
  op o : Oid Integer -> Oid .
  msg _pop : Oid -> Msg .
  msg _push_ : Oid Oid -> Msg .
  msg _top_ : Oid Oid -> Msg .
  msg element : Oid X$Elt -> Msg .
  rl < O:Oid : Stack{X}| head : O’:Oid >
     < O’:Oid : Node{X}| next : O’’:Oid >
     O:Oid pop
     => < O:Oid : Stack{X}| head : O’’:Oid >
     [label pop] .
  rl < O:Oid : Stack{X}| head : O’:Oid >
```

[3]The `including CONFIGURATION+ .` declaration in the shown module will be explained in Section 22.8.

```
    < O':Oid : Node{X}| node : O'':Oid >
    < O'':Oid : X$Cell | contents : E:X$Elt >
    O:Oid top O''':Oid
    => < O:Oid : Stack{X}| none >
       < O':Oid : Node{X}| none >
       < O'':Oid : X$Cell | none >
       element(O'':Oid,E:X$Elt)
    [label top] .
  rl < O:Oid : Stack{X}| head : null >
    O:Oid push O':Oid
    => < O:Oid : Stack{X}| head : o(O:Oid,0)>
       < o(O:Oid,0): Node{X}| next : null,node : O':Oid >
    [label push2] .
  rl < O:Oid : Stack{X}| head : o(O:Oid,N:Integer)>
    O:Oid push O':Oid
    => < O:Oid : Stack{X}| head : o(O:Oid,N:Integer + 1)>
       < o(O:Oid,N:Integer + 1): Node{X}|
           next : o(O:Oid,N:Integer),node : O':Oid >
    [label push1] .
endom
```

22.4.2 Module instantiation

We show in this section how, by instantiating the object-oriented module `OO-STACK2` given in Section 22.3.3, we can obtain a specification of a stack of banking accounts. We first specify a view `Account` from the object-oriented theory `CELL` (in Section 22.3.1) to the object-oriented module `ACCOUNT` (in Section 22.1.4).

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

Now we can do the following rewriting on the module resulting from the instantiation.

```
Maude> (rew in OO-STACK2{Account}
               * (class Account to Account,
                  class Stack{Account} to Stack{Account},
                  class Node{Account} to Node{Account},
                  attr Stack{Account} . first to head,
                  attr Account . bal to balance,
                  msg _elt_ to element,
                  sort Int to Integer) :
< 'stack : Stack{Account} | head : null >
< 'A-73728 : Account | balance : 5000 >
< 'A-06238 : Account | balance : 2000 >
< 'A-28381 : Account | balance : 15000 >
('stack push 'A-73728)
('stack push 'A-06238)
('stack push 'A-28381)
('stack top 'A-06238)
('stack pop) .)
```

```
result Configuration :
  element('A-28381,15000)
  < 'A-06238 : Account | balance : 2000 >
  < 'A-28381 : Account | balance : 15000 >
  < 'A-73728 : Account | balance : 5000 >
  < 'stack : Stack{Account}| head : o('stack, 1)>
  < o('stack, 0) : Node{Account} | next : null, node : 'A-06238 >
  < o('stack, 1) : Node{Account} |
      next : o('stack, 0), node : 'A-73728 >
```

22.5 Example: extended rent-a-car store

This section describes a variant of the rent-a-car store example in Section 22.2 in which several interesting data structures are used to store relevant information.

Let us refine the specification of a rent-a-car store presented in Section 22.2 by adding the following regulations:

10. When a rented car is returned, the deposit is used to pay the rental charges, which are calculated in accordance with the conditions at pick-up time.
11. There are three different kinds of customers: staff, occasional, and preferred.
12. Staff members and preferred customers benefit from special discounts in all rentals.
13. A customer qualifies as "preferred" when the accumulated amount of money spent in the store by the customer is above a certain threshold.

The main differences introduced by these regulations are that we need to keep the conditions at pick-up time, so that the calculations at drop-off time are correct. We also need to distinguish the three different types of customers, with the possibility of an occasional customer being promoted to preferred if he spends a given amount of money.

As an alternative approach to the one followed previously in Section 22.2, we introduce a class `Store` of rental car stores, whose attributes represent the information concerning the general parameters of such stores: the rates applicable to each type of car, the discounts for each type of customer renting each type of car, the identifiers of the customers who are suspended, the amount of money above which occasional customers are qualified as preferred, the record with the amount of money spent in the store by each of the customers, and the daily penalty for late return (20%). In addition, attributes `customers`, `cars`, `rentals`, and `calendar` store the identifiers of the objects participating in the relationships with the `Store` composite object; those are *directed* binary relationships and therefore we need only store the identifiers of the subordinate objects as attributes of the object that references them.

```
class Store |
  discounts : PFun{Tuple{Cid, Cid}, Nat},
  payments : PFun{Oid, Nat},
  penalty  : Nat,
  threshold : Nat,
  suspended : Set{Oid},
  rates    : PFun{Cid, Nat},
  customers : Set{Oid},
  cars     : Set{Oid},
  rentals  : Set{Oid},
  calendar : Oid .
```

The information on rates, discounts, and money spent is modeled by attributes of sort `PFun` of partial functions[4] (see Section 6.3.7), associating the appropriate values to each of the different agents involved. The rates for the different cars are stored in the attribute `rates`, of sort `PFun{Cid, Nat}`, that associates the per-day rate to be charged to a customer for renting a given type of car. Thus, assuming that `Rts` is a variable of sort `PFun{Cid, Nat}`, with value the partial function assigning the appropriate rates to each type of car, we have that `Rts[FullSizeCar]` is the per-day rate for renting a full size car. If we want to increase this rate by, say 20%, we can use the expression

```
Rts[FullSizeCar -> Rts[FullSizeCar] * (100 + penalty) / 100]
```

with `penalty` a constant equal to 20. The discounts applied to each customer on each type of car and the amount of the purchases of each customer are stored, respectively, in attributes `payments` and `discounts`. The set of the identifiers of the customers who are suspended is stored in an attribute `suspended` of sort `Set{Oid}`. The predefined sorts `Oid` and `Cid` are used for object identifiers and class identifiers, respectively.

This specification will allow us, for instance, to easily "compose" systems with different particular details (e.g., discounts may change from one store to another), allowing them to easily co-exist.

The rest of the classes can be specified as follows:

```
class Customer | cash : Nat, debt : Nat .
class Staff .
class OccasionalCust .
class PreferredCust .
subclasses OccasionalCust PreferredCust Staff < Customer .

class Car | available : Bool .
class EconomyCar .
class MidSizeCar .
class FullSizeCar .
subclasses EconomyCar MidSizeCar FullSizeCar < Car .

class Rental |
  deposit   : Nat, discount  : Nat,
  dueDate   : Nat, pickUpDate : Nat,
  rate      : Nat, customer  : Oid,
  car       : Oid .
```

The different actions may then be defined as follows:

```
crl [car-rental] :
  < U : Customer | cash : M >
  < I : Car | available : true >      *** the car is available
  < V : Store | suspended : US,
    rates : Rts, discounts : Dscnts, calendar : C,
    cars : (I, IS), customers : (U, SS), rentals : RS >
  < C : Calendar | date : Today >
  => < U : Customer | cash : sd(M, Amnt) >
     < I : Car | available : false >
     < V : Store | rentals : (A, RS) >
     < C : Calendar | >
     < A : Rental | pickUpDate : Today, dueDate : Today + NumDays,
```

[4] An alternative possibility is to use the maps specified in the predefined `MAP` module in Section 7.14.

```
      car : I, deposit : Amnt, customer : U,
      rate : Rt, discount : Dscnt >
   if not U in US              *** the customer is not suspended
     /\ Rt := Rts[getClass(< I : Car | >)]
     /\ Dscnt := Dscnts[(getClass(< U : Customer | >),
                        getClass(< I : Car | >))]
     /\ Amnt := sd(Rt, Dscnt) * NumDays
     /\ M >= Amnt              *** enough cash to make a deposit
   [nonexec] .
```

Notice the use of `customer` and `car` attributes in objects of class `Rental`, which makes explicit that a rental relationship is between the customer and the car specified by these attributes. Likewise for attributes `customers`, `cars`, and `calendar` of object `V` of class `Store`, which indicate that the customer, car and calendar appearing on the rule should be known to the store. After the `car-rental` action, the rental is added to the set of rentals kept by the store.

Rules may be applied to objects of the classes specified in the rules or of any of their subclasses. Remember that the function `getClass` takes an object as argument and returns its actual class (see Section 22.1.2); for example, the `getClass` function applied to an object of the form `< 'c123 : MidSizeCar | ... >` returns `MidSizeCar`, and not `Car`. Finally, notice the use of *matching equations* of the form `t := t'` in the condition (see Section 4.3).

The return of a rented car is specified by the rules below. The first rule handles the case in which the car is returned on time, that is, the current date is smaller than or equal to the due date, and therefore the deposit is greater than or equal to the amount due. Notice that the rate and discount to be used in the calculation of the amount due are those at pick-up time, which are stored as attributes of the `Rental` object.

```
crl [on-date-car-return] :
  < U : Customer | cash : M >
  < I : Car | >
  < A : Rental | customer : U, car : I, rate : Rt, discount : Dscnt,
    pickUpDate : Ppdt, dueDate : Ddt, deposit : Dpst >
  < V : Store | payments : Pmnts, cars : (I, IS),
    customers : (U, SS), calendar : C, rentals : (A, RS) >
  < C : Calendar | date : Today >
  => < U : Customer | cash : M + sd(Dpst, Amnt) >
     < I : Car | available : true >
     < V : Store | rentals : RS,
       payments : (if Pmnts[U] == undefined *** no record for customer
                   then Pmnts[U -> Amnt]
                   else Pmnts[U -> ((Pmnts[U]) + Amnt)]
                   fi) >
     < C : Calendar | >
  if (Today <= Ddt) /\ Amnt := sd(Rt, Dscnt) * sd(Today, Ppdt) .
```

In this case, the deposit is greater than the amount due and therefore part of the deposit needs to be reimbursed. Note also that the `Store` object keeps a record of the amount of money spent by each customer in the store, and thus it must be updated accordingly. We can see how the `Rental` object disappears in the righthand side of the rules: it is removed from the set of rentals known to the store and the availability of the car is restored.

The second rule deals with the case in which the car is returned late. The amount to be paid is calculated at drop-off time, but the rate and discount to be used, those at pick-up time, may have changed when returning the car.

```
crl [late-car-return] :
  < U : Customer | debt : M >
  < I : Car | >
  < A : Rental | customer : U, car : I, rate : Rt, discount : Dscnt,
    pickUpDate : Ppdt, dueDate : Ddt, deposit : Dpst >
  < V : Store | payments : Pmnts, penalty : Pnlt, rentals : (A, RS),
    cars : (I, IS), customers : (U, SS), calendar : C >
  < C : Calendar | date : Today >
  => < U : Customer | debt : M + sd(Amnt, Dpst) >
     < I : Car | available : true >
     < V : Store | rentals : RS,
       payments : (if Pmnts[U] == undefined
                   then Pmnts[U -> Dpst]
                   else Pmnts[U -> ((Pmnts[U]) + Dpst)]
                   fi) >
     < C : Calendar | >
  if Ddt < Today                        *** it is returned late
     /\ Amnt := (sd(Rt, Dscnt) * sd(Ddt, Ppdt))
          + (((sd(Rt, Dscnt) * sd(Today, Ddt))
              * (100 + Pnlt)) quo 100) .
```

In this case the customer's debt is increased by the portion of the amount due not covered by the deposit.

Debts may be paid at any time, the only condition being that the amount paid is between zero and the amount of money of the customer at that time.

```
crl [pay-debt] :
  < V : Store |
      payments : Pmnts, customers : (U, SS), calendar : C >
  < U : Customer | debt : M, cash : N >
  < C : Calendar | date : Today >
  => < V : Store | payments : Pmnts[U -> ((Pmnts[U]) + Amnt)] >
     < U : Customer | debt : sd(M, Amnt), cash : sd(N, Amnt) >
     < C : Calendar | >
  if 0 < Amnt /\ Amnt <= N /\ Amnt <= M
  [nonexec] .
```

We are assuming that, if there is a debt, then there has been a previous payment, and therefore there is already a record for that customer.

The text says that customers who are late in returning a rented car or in paying their debts "may" be suspended. However, nothing is said about the reasons for taking such a decision or when they should be suspended, that is, a customer could be suspended right after the car is returned without having paid all the charges, after some grace days, or never. In practice there will be fixed criteria, as, for example, suspending customers that are two days late, or two months.

The first rule deals with the case in which a customer has a pending debt, and the second one handles the case in which a customer is late in returning a rented car.

```
crl [suspend-late-payers] :
  < V : Store | suspended : US, customers : (U, SS) >
  < U : Customer | debt : M >
  => < V : Store | suspended : (U, US) >
     < U : Customer | >
  if (not U in US) and M > 0 .
```

```
crl [suspend-late-returns] :
  < V : Store | suspended : US, cars : (I, IS),
                customers : (U, SS), calendar : C >
  < U : Customer | >
  < I : Car | >
  < A : Rental | customer : U, car : I, dueDate : F >
  < C : Calendar | date : Today >
  => < V : Store | suspended : (U, US) >
     < U : Customer | >
     < I : Car | >
     < A : Rental | >
     < C : Calendar | >
  if (not U in US) and F < Today .
```

The upgrade in the status of a customer can then be modeled with the following rule:

```
crl [upgrade-to-preferred] :
  < U : OccasionalCust | cash : M, debt : N >
  < V : Store | threshold : Thrshld, payments : Pmnts,
                customers : (U, SS), calendar : C >
  < C : Calendar | date : Today >
  => < U : PreferredCust | cash : M, debt : N >
     < V : Store | >
     < C : Calendar | >
  if (Pmnts[U]) >= Thrshld .
```

In this rule, a customer object `U` of subclass `OccasionalCust` becomes of subclass `PreferredCust` when the accumulated amount of purchases exceeds the store's threshold. The partial function stored in the attribute `payments` gives us the amount of money spent by each customer. In Maude, objects changing their classes must show all their attributes in the righthand sides of the rules.

As in the simpler rent-a-car system, the presence of nonterminating rules and of rules with new variables in the righthand side requires some kind of strategy for the execution of the system; we give an example of such a strategy in the next section.

22.6 A strategy for sequential rule execution

Strategies are necessary for controlling the execution of rules that are not terminating, or that do not satisfy the admissibility conditions discussed in Section 5.3. A simple but interesting strategy may be one that allows us to execute a given sequence of rules, that is, to accomplish sequentially a series of actions from a particular initial state. We introduce in this section such a generic strategy and illustrate its use by applying it for executing the systems specified in Sections 22.2 and 22.5. Dealing with strategies may become cumbersome, since we need to handle terms and modules at different levels of reflection, and expressions may become quite hard to read and handle. We show in this section how the `upModule` and `upTerm` functions and the `down` command introduced in Section 21.4 can help in alleviating this difficulty.

A strategy is represented as a sequence of rule applications. We instantiate the predefined module `LIST` with pairs formed by a rule label representing the rule to be applied, and a substitution to partially instantiate the variables in such a rule before its application. The pairs are obtained using the generic tuple construction described in Section 21.3.1. Thus, to get the module expression `LIST{Tuple{Qid, Substitution}}`, given the predefined view `Qid` and the *parameterized view* `Tuple`, that we have already used in the partial functions example of Section 6.3.7, we only need to define a view `Substitution` from `TRIV` to `META-LEVEL`.

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

This construction is put to work in the module `REW-SEQ` below. The operator `rewSeq` in this module takes the metarepresentation of a module, the metarepresentation of a term, and a list of pairs (each formed by a rule label and a substitution); the term obtained in this way is rewritten by applying the given rules sequentially, using in their applications their corresponding partial substitutions.

```
(mod REW-SEQ is
  including META-LEVEL .
  protecting LIST{Tuple{Qid, Substitution}} .

  var M : Module .
  var T : Term .
  var L : Qid .
  var S : Substitution .
  var LLS : List{Tuple{Qid, Substitution}} .

  op rewSeq :
      Module Term List{Tuple{Qid, Substitution}} -> [Term] .

  rl [seq] : rewSeq(M, T, (L, S) LLS)
    => rewSeq(M,
         getTerm(metaXapply(M, T, L, S, 0, unbounded, 0)), LLS) .

  rl [seq] : rewSeq(M, T, nil) => T .
endm)
```

The rules to be applied here are part of the module given as first argument. The strategy starts with the term given as initial state, which is replaced in each recursive call by the term representing the state obtained after the application of the next rule in the sequence (see Section 17.6.4). When all the rules have been applied, thus reaching the empty list as third argument, the current state is returned as the resulting final state.

We illustrate the use of the `rewSeq` strategy by applying a sequence of rules on a configuration of the rent-a-car system specified in Section 22.2. Let `RENT-A-CAR-STORE` be the name of the module containing the specification of such a system, and let `StoreConf` be a configuration of objects defined in the following module.

```
(fmod RENT-A-CAR-STORE-TEST is
  pr RENT-A-CAR-STORE .

  op StoreConf : -> Configuration [memo] .
  eq StoreConf
    = < 'C1 : Customer | cash : 5000, debt : 0, suspended : false >
      < 'C2 : Customer | cash : 5000, debt : 0, suspended : false >
      < 'A1 : EconomyCar | available : true, rate : 100 >
      < 'A3 : MidSizeCar | available : true, rate : 150 >
      < 'A5 : FullSizeCar | available : true, rate : 200 >
      < 'C : Calendar | date : 0 > .
endfm)
```

The `StoreConf` configuration consists of two clients `C1` and `C2`, three cars `A1`, `A3` and `A5`, and a calendar object `C`. Now, let `StoreStrat` be a sequence of pairs (rule label - substitution) that

defines the strategy declared in the following module as a sequence of actions:

```
(fmod REW-SEQ-TEST is
  pr REW-SEQ .

  op StoreStrat : -> List{Tuple{Qid, Substitution}} [memo] .
  eq StoreStrat
    = ('car-rental,
       'U:Oid <- ''C1.Qid ;        *** size car A3 for 2 days
       'I:Oid <- ''A3.Qid ;
       'NumDays:Int <- 's_^2['0.Zero] ;
       'A:Oid <- ''a0.Qid)
      ('new-day, none)             *** two days pass
      ('new-day, none)
      ('on-date-car-return, none)  *** car A3 is returned
      ('new-day, none)
      ('car-rental,                *** client C1 rents the full
       'U:Oid <- ''C1.Qid ;        *** size car A5 for 1 day
       'I:Oid <- ''A5.Qid ;
       'NumDays:Int <- 's_^1['0.Zero] ;
       'A:Oid <- ''a1.Qid)
      ('new-day, none)             *** two days pass
      ('new-day, none)
      ('late-car-return, none)     *** car A5 is returned
      ('new-day, none)
      ('suspend-late-payers, none) *** client C1 is suspended
      ('new-day, none)
      ('new-day, none)
      ('pay-debt,                  *** client C1 pays 100$
       'Amnt:Int <- 's_^100['0.Zero]) .
endfm)
```

Comments on the righthand side of the code above explain the sequence of rules defining the strategy. Basically, the execution trace specified consists of client `C1` renting two cars, one of which is returned on time and the other one is returned late. After the second car is returned, the client is suspended for being late in his payments. The client then pays part of his debt. Note how the passage of time is modeled by the application of the rule `new-day`.

Now, in order to execute the system specifications using this strategy, we just need to use `rewSeq` to apply the given rules sequentially, using their corresponding partial substitutions in their applications. Note how the first two arguments are metarepresented with the `upModule` and `upTerm` functions, since they need to be the metarepresentations of the actual module and term, respectively.

```
Maude> (down RENT-A-CAR-STORE :
         rew rewSeq(upModule(RENT-A-CAR-STORE-TEST),
           upTerm(RENT-A-CAR-STORE-TEST, StoreConf),
           StoreStrat) .)

result Configuration :
  < 'C : Calendar | date : 8 >
  < 'C1 : Customer | suspended : true, debt : 140, cash : 4400 >
  < 'C2 : Customer | suspended : false, debt : 0, cash : 5000 >
  < 'A1 : EconomyCar | rate : 100, available : true >
  < 'A3 : MidSizeCar | rate : 150, available : true >
  < 'A5 : FullSizeCar | rate : 200, available : true >
```

We can see in this configuration that eight days have passed, after which the client C1 is suspended. The client C1 has paid a total of \$600 ($= 2 \times 150 + 200 + 100$), and has still a debt of \$140 ($= 200 + 20\ \%\ 200 - 100$).

The same strategy can be used to execute the extended specification in Section 22.5, contained in a module named EXTENDED-RENT-A-CAR-STORE. First, we define a module with an initial configuration ExtStoreConf.

```
(fmod EXTENDED-RENT-A-CAR-STORE-TEST is
  pr EXTENDED-RENT-A-CAR-STORE .

  op ExtStoreConf : -> Configuration [memo] .
  eq ExtStoreConf
    = < 'S : Store |
        discounts :
          (((Staff, EconomyCar), 20),
           ((Staff, MidSizeCar), 30),
           ((Staff, FullSizeCar), 40),
           ((OccasionalCust, EconomyCar), 0),
           ((OccasionalCust, MidSizeCar), 0),
           ((OccasionalCust, FullSizeCar), 0),
           ((PreferredCust, EconomyCar), 10),
           ((PreferredCust, MidSizeCar), 15),
           ((PreferredCust, FullSizeCar), 20)),
        payments : empty, penalty : 0,
        threshold : 1000, suspended : empty,
        rates : ((EconomyCar, 100),
                 (MidSizeCar, 150),
                 (FullSizeCar, 200)),
        customers : ('C1, 'C2),
        cars : ('A1, 'A3, 'A5),
        rentals : empty, calendar : 'C >
      < 'C1 : Staff | cash : 5000, debt : 0 >
      < 'C2 : OccasionalCust | cash : 5000, debt : 0 >
      < 'A1 : EconomyCar | available : true >
      < 'A3 : MidSizeCar | available : true >
      < 'A5 : FullSizeCar | available : true >
      < 'C : Calendar | date : 0 > .
endfm)
```

Now we execute a command completely analogous to the previous one, obtaining a resulting state that shows how, after eight days, client C1 has paid \$500, and has a debt of \$60.

```
Maude> (down EXTENDED-RENT-A-CAR-STORE :
          rew rewSeq(upModule(EXTENDED-RENT-A-CAR-STORE),
               upTerm(EXTENDED-RENT-A-CAR-STORE, ExtStoreConf),
               StoreStrat) .)
result Configuration :
  < 'A1 : EconomyCar | available : true >
  < 'A3 : MidSizeCar | available : true >
  < 'A5 : FullSizeCar | available : true >
  < 'C : Calendar | date : 8 >
  < 'C1 : Staff | cash : 4500, debt : 60 >
  < 'C2 : OccasionalCust | cash : 5000, debt : 0 >
  < 'S : Store | calendar : 'C,
     cars : ('A1, 'A3, 'A5),
```

```
    customers : ('C1, 'C2),
    discounts : (((OccasionalCust, EconomyCar), 0),
                 ((OccasionalCust, FullSizeCar), 0),
                 ((OccasionalCust, MidSizeCar), 0),
                 ((PreferredCust, EconomyCar), 10),
                 ((PreferredCust, FullSizeCar), 20),
                 ((PreferredCust, MidSizeCar), 15),
                 ((Staff, EconomyCar), 20),
                 ((Staff, FullSizeCar), 40),
                 ((Staff, MidSizeCar), 30)),
    payments : ('C1, 500),
    penalty : 0,
    rates : ((EconomyCar, 100),
             (FullSizeCar, 200),
             (MidSizeCar, 150)),
    rentals : empty,
    suspended : 'C1,
    threshold : 1000 >
```

22.7 Model checking a round-robin scheduling algorithm

In this section we present a specification of a round-robin scheduling algorithm, and the mutual exclusion and guaranteed reentrance properties are proven about it. Both the algorithm and the property guaranteing that all processes reenter their critical sections are parameterized by the number of processes. We use Maude's model checker to prove the mutual exclusion and guaranteed reentrance properties. As we said in Section 21.2, to use the `MODEL-CHECKER` module, or any other Core Maude module, we just need to make sure that it has been loaded; we suggest loading the `model-checker.maude` file *before* starting Full Maude.

We first give a specification of natural numbers modulo. Since we want to be able to have any number of processes, we define the `NAT/` module parameterized by the functional theory `NZNAT#`, which requires a constant of sort `Nat`. Thus, having a view, say `5` from `TRIV` to `NZNAT#` mapping `#` to the natural number `5`, the module expression `NAT/{5}` specifies the natural numbers modulo 5.

```
(fth NZNAT# is
  protecting NAT .
  op # : -> NzNat .
endfth)

(fmod NAT/{N :: NZNAT#} is
  sort Nat/{N} .
  op `[_`] : Nat -> Nat/{N} [ctor] .
  op _+_ : Nat/{N} Nat/{N} -> Nat/{N} .
  op _*_ : Nat/{N} Nat/{N} -> Nat/{N} .
  vars N M : Nat .
  ceq [N] = [N rem #] if N >= # .
  eq [N] + [M] = [N + M] .
  eq [N] * [M] = [N * M] .
endfm)
```

The round-robin scheduling algorithm is specified in the module `RROBIN` below. Processes are represented as objects of class `Proc`, which may be in `wait` or `critical` mode, meaning

that a process may be either in its critical section or waiting to enter into it. The process getting the token, which is represented as the message `go`, can enter its critical section. Once a process gets out of its critical section it forwards the token to the next process. The `init` operator sets up the initial configuration for a given number of processes. Note that `Nat/{N}` is made a subsort of `Oid`, making in this way natural numbers modulo `N` valid object identifiers.

```
(omod RROBIN{N :: NZNAT#} is
  protecting NAT/{N} .

  sort Mode .
  ops wait critical : -> Mode [ctor] .

  subsort Nat/{N} < Oid .

  class Proc | mode : Mode .
  msg go : Nat/{N} -> Msg .

  var N : Nat .

  rl [enter] :
    go([N])
    < [N] : Proc | mode : wait >
    => < [N] : Proc | mode : critical > .

  rl [exit] :
    < [N] : Proc | mode : critical >
    => < [N] : Proc | mode : wait >
       go([s(N)]) .

  op init : -> Configuration .
  op make-init : Nat/{N} -> Configuration .

  ceq init = go([0]) make-init([N]) if s(N) := # .
  ceq make-init([s(N)])
    = < [s(N)] : Proc | mode : wait > make-init([N])
    if N < # .
  eq make-init([0]) = < [0] : Proc | mode : wait > .
endom)
```

For proving mutual exclusion and guaranteed reentrance, we declare the propositions `inCrit` and `twoInCrit` in the module `CHECK-RROBIN` below (see Chapter 12 for a discussion on the use of Maude's model checker). The property `inCrit` takes a `Nat/{N}` as argument, thus making this property parameterized by the number of processes, and is true when such a process is in its critical section. The property `twoInCrit` is true if any two processes are in their critical sections simultaneously. Mutual exclusion will be proved directly below, while for proving guaranteed reentrance we use the auxiliary formula `guaranteedReentrance`, which allows us to specify the property of all processes reentering their critical sections in exactly $2N$ steps, for N the number of processes. For a formula `F`, `nextIter F` returns `O...O F` (where `O` denotes the modal next operator), which specifies that the property is true in the next iteration, that is, $2N$ steps later. Note that the expression `2 * #` will become two times N once the module is instantiated.

```
(omod CHECK-RROBIN{N :: NZNAT#} is
   pr RROBIN{N} .
```

```
  inc MODEL-CHECKER .
  inc SATISFACTION .
  ex LTL-SIMPLIFIER .
  inc LTL .

  subsort Configuration < State .

  op inCrit : Nat/{N} -> Prop .
  op twoInCrit : -> Prop .

  var  N : Nat .
  vars X Y : Nat/{N} .
  var  C : Configuration .
  var  F : Formula .

  eq < X : Proc | mode : critical > C |= inCrit(X) = true .
  eq < Y : Proc | mode : critical > < Y : Proc | mode : critical > C
       |= twoInCrit = true .

  op guaranteedReentrance : -> Formula .
  op allProcessesReenter : Nat -> Formula .
  op nextIter_ : Formula -> Formula .
  op nextIterAux : Nat Formula -> Formula .

  eq guaranteedReentrance = allProcessesReenter(#) .

  eq allProcessesReenter(s N)
    = (inCrit([s N]) -> nextIter inCrit([s N])) /\
      allProcessesReenter(N) .
  eq allProcessesReenter(0) = inCrit([0]) -> nextIter inCrit([0]) .

  eq nextIter F = nextIterAux(2 * #, F) .

  eq nextIterAux(s N, F) = O nextIterAux(N, F) .
  eq nextIterAux(0, F) = F .
endom)
```

Note that the LTL formula describing the `guaranteedReentrance` property is not a single LTL formula, but an *infinite parametric family of formulas*

$$\texttt{guaranteedReentrance} = \{\texttt{allProcessesReenter}(n) \mid n \in \mathbb{N}\}.$$

The use of equations in the above `CHECK-RROBIN` parameterized module allows us to define this infinite family of formulas by means of a few recursive equations. When this module is instantiated for a concrete value of n, we then obtain the concrete LTL formula `allProcessesReenter(`n`)` for that n.

We now prove mutual exclusion and guaranteed reentrance for the case of five processes using the model checker.

```
(view 5 from NZNAT# to NAT is
  op # to term 5 .
endv)

Maude> (reduce in CHECK-RROBIN{5} :
         modelCheck(init, [] ~ twoInCrit) .)
```

```
result Bool :
  true

Maude> (reduce in CHECK-RROBIN{5} :
          modelCheck(init, [] guaranteedReentrance) .)
result Bool :
  true
```

Of course the answer depends on the property checked and is not always `true`. The following example shows how the model checker gives a counterexample as result when trying to prove that, for a configuration of five processes, process `[1]` is in its critical section three steps after it was in it.

```
Maude> (red in CHECK-RROBIN{5} :
          modelCheck(init, [] (inCrit([1]) -> O O O inCrit([1]))) .)
result ModelCheckResult :
  counterexample(
    {go([0]) <[0]: Proc | mode : wait >
     <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
     <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
    {<[0]: Proc | mode : critical > <[1]: Proc | mode : wait >
     <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >
     <[4]: Proc | mode : wait >, 'exit}
    {go([1]) <[0]: Proc | mode : wait >
     <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
     <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
    {<[0]: Proc | mode : wait > <[1]: Proc | mode : critical >
     <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >
     <[4]: Proc | mode : wait >, 'exit}
    {go([2]) <[0]: Proc | mode : wait >
     <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
     <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
    {<[0]: Proc | mode : wait > <[1]: Proc | mode : wait >
     <[2]: Proc | mode : critical > <[3]: Proc | mode : wait >
     <[4]: Proc | mode : wait >, 'exit},
    {go([3]) <[0]: Proc | mode : wait >
     <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
     <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
    {<[0]: Proc | mode : wait > <[1]: Proc | mode : wait >
     <[2]: Proc | mode : wait > <[3]: Proc | mode : critical >
     <[4]: Proc | mode : wait >, 'exit}
    {go([4]) <[0]: Proc | mode : wait >
     <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
     <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
    {<[0]: Proc | mode : wait > <[1]: Proc | mode : wait >
     <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >
     <[4]: Proc | mode : critical >, 'exit}
    {go([0]) <[0]: Proc | mode : wait >
     <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
     <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
    {<[0]: Proc | mode : critical > <[1]: Proc | mode : wait >
     <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >
     <[4]: Proc | mode : wait >, 'exit}
    {go([1]) <[0]: Proc | mode : wait >
     <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
```

```
 <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
{<[0]: Proc | mode : wait > <[1]: Proc | mode : critical >
 <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >
 <[4]: Proc | mode : wait >, 'exit}
{go([2]) <[0]: Proc | mode : wait >
 <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
 <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
{<[0]: Proc | mode : wait > <[1]: Proc | mode : wait >
 <[2]: Proc | mode : critical > <[3]: Proc | mode : wait >
 <[4]: Proc | mode : wait >, 'exit})
```

22.8 From object-oriented modules to system modules

The best way to understand classes and class inheritance in Full Maude is by making explicit the full structure of an object-oriented module, which is left somewhat implicit by the syntactic conventions adopted for them. Indeed, although object-oriented modules provide convenient syntax for programming object-oriented systems, their semantics can be reduced to that of system modules. 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.

In the translation process, the most basic structure shared by all object-oriented modules is made explicit by the `CONFIGURATION` functional module. The translation of a given object-oriented module extends this structure with the classes, messages and rules introduced by the module. For example, the following system module is the translation of the `ACCOUNT` module introduced earlier. Note that a subsort `Account` of `Cid` is introduced. The purpose of this subsort is to range over the class identifiers of the subclasses of `Account`. For the moment, no such subclasses have been introduced; therefore, at present the only constant of sort `Account` is the class identifier `Account`.

```
mod ACCOUNT is
  protecting INT .
  protecting QID .
  including CONFIGURATION+ .
  including CONFIGURATION .
  sorts Account .
  subsort Qid < Oid .
  subsort Account < Cid .
  op Account : -> Account .
  op credit : Oid Int -> Msg [msg] .
  op debit : Oid Int -> Msg [msg] .
  op from_to_transfer_ : Oid Oid Int -> Msg [msg] .
  op bal :_ : Int -> Attribute .
  var A : Oid .
  var B : Oid .
  var M : Int .
  var N : Int .
  var N' : Int .
  var V@Account : Account .
  var ATTS@0 : AttributeSet .
  var V@Account1 : Account .
  var ATTS@2 : AttributeSet .
  rl [credit] :
```

```
    credit(A, M)
    < A : V@Account | bal : N, ATTS@0 >
    => < A : V@Account | bal : (N + M), ATTS@0 > .
  crl [debit] :
    debit(A, M)
    < A : V@Account | bal : N, ATTS@0 >
    => < A : V@Account | bal : (N - M), ATTS@0 >
    if N >= M = true .
  crl [transfer] :
    (from A to B transfer M)
    < A : V@Account | bal : N, ATTS@0 >
    < B : V@Account1 | bal : N', ATTS@2 >
    => < A : V@Account | bal : (N - M), ATTS@0 >
       < B : V@Account1 | bal : (N' + M), ATTS@2 >
    if N >= M = true .
endm
```

We can describe the desired transformation from an object-oriented module to a system module as follows:[5]

- The module `CONFIGURATION+` is imported, which in turn imports the module `CONFIGURATION` from Section 8.1. It adds a function `getClass` which returns the actual class of the given object, and also syntax for objects with no attributes `<_:_| >`.

  ```
  mod CONFIGURATION+ is
    protecting CONFIGURATION .
    op <_:_| > : Oid Cid -> Object .
    op class : Object -> Cid .
    eq < O:Oid : C:Cid | > = < O:Oid : C:Cid | none > .
    eq getClass(< O:Oid : C:Cid | A >) = C:Cid .
  endm
  ```

- For each class declaration of the form `class` C `|` a_1`:`S_1`,` $\ldots$ `,`a_n`:`S_n, the following is introduced: a subsort C of sort `Cid`, a constant C of sort C, and declarations of operations

  $$a_i \texttt{ :\_ : } S_i \texttt{ -> Attribute}$$

  for each attribute a_i.

- For each subclass relation C `<` D a subsort declaration

  `subsort` C `<` D `.`

  is introduced, and the set of attributes for objects of class C is completed with those of D.

- The system modules resulting from the transformation have the special features supported in Core Maude for object-based programming explained in Chapter 8. Specifically, the `msg` attribute is added to message declarations starting with the `msg` keyword.

- The rewrite rules are modified to make them applicable to all objects of the given classes and of their subclasses, that is, not only to objects whose class identifiers are those explicitly given. The rules are then "inherited" by all objects in their subclasses by replacing

[5] We have simplified the transformation of object-oriented modules into system modules that originally appeared in [104].

the class identifiers in the objects in the rules by variables of the corresponding class sort. Variables of sort `AttributeSet` are also introduced, to range over the additional attributes that may appear in objects of a subclass. That is, each object expression `< O : C | ... >` appearing in a rule is translated into `< O : X | ..., Atts >`, where the new variable X is declared of sort C, and the new variable *Atts* has sort `AttributeSet`.

- The rewrite rules are modified to give the user the possibility of not mentioning in a given rule those attributes of an object that are not relevant for that rule. To explain the transformation, let $\overrightarrow{a : v}$ denote the attribute-value pairs $a_1 : v_1, \ldots, a_n : v_n$, where $\overrightarrow{a}$ are the attribute identifiers of a given class C (after completing it with all the attributes in its superclasses) having $\overrightarrow{S}$ as the corresponding sorts of values prescribed for those attributes. Then, in object-oriented modules we allow rules where the attributes for an object O, mentioned in the lefthand and righthand sides of a rule, need not exhaust all the object's attributes, but can instead be in any of two arbitrary subsets of the object's attributes. We can picture this as follows

$$\ldots \langle O : C \mid \overrightarrow{al : vl}, \overrightarrow{ab : vb} \rangle \ldots \longrightarrow \ldots \langle O : C \mid \overrightarrow{ab : vb'}, \overrightarrow{ar : vr} \rangle \ldots$$

  where $\overrightarrow{al}$ are the attributes appearing only on the *left*, $\overrightarrow{ab}$ are the attributes appearing on *both* sides, and $\overrightarrow{ar}$ are the attributes appearing only on the *right*. In the transformation into a system module, this rule is translated into

$$\begin{aligned} &\ldots \langle O : X \mid \overrightarrow{al : vl}, \overrightarrow{ab : vb}, \overrightarrow{ar : x}, \overrightarrow{ac : x'}, Atts \rangle \ldots \\ &\quad \longrightarrow \ldots \langle O : X \mid \overrightarrow{al : vl}, \overrightarrow{ab : vb'}, \overrightarrow{ar : vr}, \overrightarrow{ac : x'}, Atts \rangle \ldots \end{aligned}$$

  where X is a variable of sort C, $\overrightarrow{ac}$ are the attributes defined in the class C that do not appear in $\overrightarrow{al}$, $\overrightarrow{ab}$, or $\overrightarrow{ar}$, the $\overrightarrow{x}$ and $\overrightarrow{x'}$ are new variables of the appropriate sorts, and *Atts* matches the remaining attribute-value pairs.

The rewrite rules given in the original `ACCOUNT` module are interpreted here—according to the conventions already explained—in a form that can be inherited by subclasses of `Account` that could be defined later. Thus, `SavingAccount` inherits the rewrite rules for crediting and debiting accounts, and for transferring funds between accounts that had been defined for `Account`.

Let us illustrate the treatment of class inheritance with the system module resulting from the transformation of the module `SAVING-ACCOUNT` introduced previously.

```
mod SAVING-ACCOUNT is
  including CONFIGURATION+ .
  including CONFIGURATION .
  including ACCOUNT .
  sorts SavingAccount .
  subsort SavingAccount < Cid .
  subsort SavingAccount < Account .
  op SavingAccount : -> SavingAccount .
  op rate :_ : Int -> Attribute .
endm
```

Note that by translating a rule like `credit` above

```
rl [credit] :
```

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

into its corresponding transformed form

```
rl [credit] :
  credit(A, M)
  < A : V0@:Account | bal : N, V1@:AttributeSet >
  => < A : V0@:Account | bal : (N + M), V1@:AttributeSet > .
```

it is guaranteed that the rule will be applicable to objects of class `Account` as well as of any of its subclasses.

Note also that a rule like `change-age` (discussed in Section 22.1.4)

```
rl [change-age] :
  < O : Person | >
  to O : new age A
  => < O : Person | age : A > .
```

is translated into a rule like

```
rl [change-age] :
  < O : V0@:Person | name : V1:String, age : V2:Nat,
                    account : V3:Oid, V4@:AttributeSet >
  to O : new age A
  => < O : V0@:Person | age : A, name : V1:String,
                        account : V3:Oid, V4@:AttributeSet > .
```

With this translation we allow the rule to be applied to objects in subclasses of `Person`. Furthermore, we guarantee that it is only applied to well-formed objects, that is, to objects with all the required attributes.

See [44] for a detailed explanation of the transformation of object-oriented modules into system modules and how their semantics *is* by definition that of the original object-oriented module.

Part III

Reference

Chapter 23

Complete List of Maude Commands

In this chapter we use curly bracket pairs, '{' and '}', to enclose optional syntax.

23.1 Command line flags

The following command line flags are supported.

--help
Displays information on the usage of the Maude command and its line flags.

--version
Displays the Maude version number.

-no-mixfix
Turns off mixfix printing; useful if Maude is being run by some other program that does not want to deal with the intricacies of mixfix parsing.

-ansi-color, -no-ansi-color
By default ANSI escape codes for color and other effects are disabled if the standard output is not a terminal or the `TERM` environment variable is set to `dumb`. These flags allow the default behavior to be overridden.

-tecla, -no-tecla
By default Tecla-based command line editing is disabled if the standard output is not a terminal or the `TERM` environment variable is set to `dumb` or `emacs`. These flags allow the default behavior to be overridden.

-no-prelude
Causes Maude not to read in the standard prelude.

-batch
Disables control-C handling.

-interactive
Pretends to be interactive, and enables control-C handling even though standard output is not a terminal.

-xml-log=*file-name*
Generates an XML log for selected commands in the given file.

-no-banner
Causes Maude not to show the welcome banner at start up.

-random-seed=*number*
Specifies the natural number *number* in the range $[0, 2^{32} - 1]$ as the seed for the pseudo-random number generator `random` in module `RANDOM` (see Section 7.3). The default seed is 0.

-no-advise
Switches off advisories at start up.

-always-advise
Disables the possibility of turning advisories off.

-no-wrap
Disables the automatic line wrapping of output.

-print-to-stderr
Causes the output of the print attribute to be set to stderr rather than stdout.

-erewrite-loop-mode
Causes Maude to use external object rewriting for loop mode.

-show-pid
Print process id to stderr before printing banner.

-allow-processes
Allow running arbitrary executables.

-allow-files
Allow operations on files.

-trust
Allow all potentially risky capabilities.

23.2 Rewriting commands

reduce {**in** ***module*** **:**} ***term*** **.**
Causes the specified term to be reduced using the equations and membership axioms in the given module. **reduce** may be abbreviated to **red**. If the **in** clause is omitted, the current module is assumed. For examples, see Section 4.9.

rewrite {[***bound***]} {**in** ***module*** **:**} ***term*** **.**
Causes the specified term to be rewritten using the rules, equations, and membership

axioms in the given module. The default interpreter for rules applies them using a rule-fair top-down (lazy) strategy and stops when the number of rule applications reaches the given bound. No rule will be applied if an equation can be applied. If the **in** clause is omitted, the current module is assumed. If the upper bound clause is omitted, infinity is assumed. **rewrite** may be abbreviated to **rew**. For examples, see Section 5.4.1.

frewrite {[***bound*** {**,*number***}]} {**in** ***module*** **:**} ***term*** **.**

Like the previous command, causes the specified term to be rewritten using the rules, equations, and membership axioms in the given module. But now the default interpreter for rules applies them using a rule- and position-fair strategy and stops when the number of rule applications reaches the given bound. This strategy causes multiple passes over the term, with at most *number* rule rewrites taking place at each position. If the **in** clause is omitted, the current module is assumed. If the upper bound clause is omitted, infinity is assumed. If the number of rewrites per position is omitted, 1 is assumed. **frewrite** may be abbreviated to **frew**. For examples, see Section 5.4.2.

Unlike `rewrite`, which uses a leftmost outermost strategy for applying rules and reduces the whole term with equations after each successful rule rewrite, `frewrite` attempts to be position fair by making a number of depth-first traversals of the term. On each traversal, each position that existed at the start of the traversal is entitled to at most *number* rule rewrites when its turn comes around. After a rule rewrite succeeds, only the subterm that was rewritten is reduced with equations in order to avoid destroying positions that have not yet had their turn for the current traversal. Traversals are made until *bound* rule rewrites have been made, or until no more rewrites are possible. When operators have the `assoc` or `iter` attributes, term depth and positions are relative to the flattened or compact form of the term, respectively. Thus, fair rewriting treats a whole stack of an `iter` operator as a single position for the purposes of position fairness.

The are a couple of caveats with `frewrite`:

- If position-fair rewriting stops in mid traversal, then the sort of the (incompletely reduced) result has not yet been calculated and is printed as `(sort not calculated)`.
- Position-fair rewriting is not substitution fair; this is particularly apparent if you have a multiset of messages and objects, as in Section 8.2.

erewrite {[***bound*** {**,*number***}]} {**in** ***module*** **:**} ***term*** **.**

Works like the `frewrite` command and in addition it allows messages to be exchanged with external objects that do not reside in the configuration. It is abbreviated to `erew`.

continue {***number***} **.**

Attempts to continue rewriting the result of the last rewriting command using the rules, equations, and membership axioms, stopping if the upper bound on the number of rule applications is reached. This command is only usable if the current module has not changed since the last rewriting command, and the last rewriting command was not `reduce`. If no upper bound clause is given, infinity is assumed. In the case where the last rewriting command was `frewrite`, the number given to the continue command increases the bound on the number of traversals, leaving the number of rewrites per position unchanged. In particular, considerable extra information about the current traversal is saved by the `frewrite` command so that, for example,

```
frewrite [n, k] t .
continue m .
```

produces the same final answer as

```
frewrite [s, k] t .
```

when `s = n + m`. For an `erewrite` command, the same state information is preserved as for `frewrite`, but in this case nothing can be guaranteed, due to the interaction with the external environment.

loop {**in** *module* **:**} *term* **.** **(deprecated)**

This command is used to initialize the read-eval-print loop in a module importing `LOOP-MODE` (see Section 18.4). The specified term is rewritten as far as possible using the rules, equations, and membership axioms in the given module. If the result has a loop constructor symbol at the top, then it becomes the current state of the loop; also, the list of quoted identifiers in the output position of the loop constructor is printed as a sequence of identifiers.

(*identifier** **)** **(deprecated)**

This command is used to input a list of identifiers to the loop in a module importing `LOOP-MODE` (see Section 18.4). If the current module has not changed since the last rewriting command, the result of previous rewrites has a loop constructor symbol at the top, and the last rewriting command was not **reduce** then:

1. the sequence of identifiers in the parentheses is converted into a list of quoted identifiers and is placed under the input position of the loop constructor;
2. a nil list of quoted identifiers is placed under the output position of the loop constructor;
3. the new term is rewritten as far as possible using the rules, equations, and membership axioms in the module to which the term belongs; and
4. if the new result has a loop constructor symbol at the top, the list of quoted identifiers in the output position of the loop constructor is printed as a sequence of identifiers.

set clear rules on . / set clear rules off .

Normally, each `rewrite` or `frewrite` command and each loop mode invocation resets the rule state for each symbol. For most symbols the rule state consists of the next rule to be executed in a round-robin scheme but for counter symbols the rule state consists of the next number to rewrite to. Setting *clear rules* to off means the rule state will *not* be reset between commands.

23.3 Matching commands

Matching commands are used to directly invoke the rewriting engine's term pattern matcher. They can be useful for figuring out exactly what subjects can be matched by a complex pattern.

match {[*number*]} {**in** *module* **:**} *pattern* `<=?` *subject-term* {**such that** *condition*} **.**

Performs straightforward matching in the given module. This kind of matching is used by the engine for applying membership axioms. The result is a list of at most *number* matching substitutions such that the subject term matches the pattern and the substitution satisfies the optional *condition* (whose syntactic form is the same as the one of conditions for conditional equations and memberships; see Section 4.3). If the upper bound clause is omitted, infinity is assumed. For examples, see Section 4.9.

xmatch {[***number***]} {**in** ***module*** **:**} ***pattern*** `<=?` ***subject-term*** {**such that** ***condition***} **.**

Works similarly to the previous command, except that it performs matching with extension for those theories that need it (those including the `assoc` or `iter` attributes). If the subject term (after theory normalization) has a symbol f from an extension theory on top, only a piece of the top theory layer with f on top need be matched. This kind of matching is used by the engine for applying equations and rules in order to accurately simulate equivalence class rewriting. The result is a list of all matches satisfying the given condition. If only part of the subject was matched, that part is given. For examples, see Sections 4.8 and 4.9.

23.4 Searching commands

search {[***bound*** {**,*depth***}]} {**in** ***module*** **:**} ***subject searchtype pattern*** {**such that** ***condition***} **.**

Performs a breadth-first search for rewrite proofs starting at *subject* to a final state that matches *pattern* and satisfies an optional *condition* (whose syntactic form is the same as the one of conditions for conditional equations and memberships; see Section 4.3). Possible values for *searchtype* are

`=>1`	one step proof
`=>+`	one or more steps proof
`=>*`	zero or more steps proof
`=>!`	only canonical final states, that cannot be further rewritten, are allowed as solutions

The optional *bound* argument provides an upper bound in the number of solutions to be found; if it is omitted, infinity is assumed.

The optional *depth* argument indicates the maximum depth of the search. If it is omitted, infinity is assumed. It is also possible to give a depth bound without giving a bound on the number of solutions returned by requesting a search of the form `search [,m] ...`.

The search type `=>1` is an abbreviation of the search type `=>+` with the depth bound set to `1`.

As usual, if the **in** clause is omitted, the current module is assumed.

For examples, see Section 5.4.3.

show search graph .

Displays the search graph generated by the last search.

show path *number* .

Displays the path to a given state, identified by the number, in a search graph.

show path labels *number* .

Works like the command above, but only shows labels of applied rules instead of the full path.

23.5 Strategic rewriting commands

srewrite {[***bound***]} {**in** ***module*** **:**} ***subject*** **by** ***strategyexpr*** **.**
Causes the specified term to be rewritten according to the given strategy in the given module. The command performs an exhaustive fair search for all the strategy solutions, unless the optional bound on the number of solutions is specified. If the **in** clause is omitted, the current module is assumed. **srewrite** may be abbreviated to **srew**.

For examples, see Section 10.1.

dsrewrite {[***bound***]} {**in** ***module*** **:**} ***subject*** **by** ***strategyexpr*** **.**
Like the previous command, but solutions are searched depth-first. **dsrewite** may be abbreviated to **dsrew**.

For more details, see Section 10.4.

23.6 Unification, variants, and narrowing commands

{***irredundant***} **unify** {[***bound***]} {**in** ***module*** **:**} $term_1$ **=?** ***term'***$_1$ { /\ ... /\ $term_k$ **=?** ***term'***$_k$ } **.**
Computes a complete set of order-sorted unifiers modulo the (supported) equational axioms in the given module for the provided unification problem. If the cardinality of the set of unifiers is greater than the specified *bound*, the unifiers beyond that bound are omitted. The *module* can be any module or theory declared in the current Maude session; as usual, if the **in** clause is omitted, the current module is used. If the keyword `irredundant` is used, a minimal set of unifiers is returned.

For examples, see Section 13.4.

{***filtered***} **variant unify** {[***bound***]} {**in** ***module*** **:**} $term_1$ **=?** ***term'***$_1$ { /\ ... /\ $term_k$ **=?** ***term'***$_k$ } {**such that** ***term"***$_1$**,** ..., ***term"***$_n$ ***irreducible***}**.**
Computes a complete set of order-sorted unifiers modulo the equations declared with the `variant` attribute (which must satisfy the finite variant property) plus the (supported) equational axioms in the given module for the provided unification problem. It may be necessary to satisfy an optional irreducibility condition on extra terms *term"*$_1$, ..., *term"*$_n$. If the cardinality of the set of unifiers is greater than the specified *bound*, the unifiers beyond that bound are omitted. The *module* can be any module or theory declared in the current Maude session; as usual, if the **in** clause is omitted, the current module is used. If the keyword `filtered` is used, a minimal set of unifiers is returned.

For more details, see Section 14.9.

variant match {[***bound***]} {**in** ***module*** **:**} $term_1$ <=? ***term'***$_1$ { /\ ... /\ $term_k$ <=? ***term'***$_k$ } {**such that** ***term"***$_1$**,** ..., ***term"***$_n$ ***irreducible***} **.**
Computes a complete set of order-sorted matches modulo the equations declared with the `variant` attribute (which must satisfy the finite variant property) plus the (supported) equational axioms in the given module for the provided matching problem. It may be necessary to satisfy an optional irreducibility condition on extra terms *term"*$_1$, ..., *term"*$_n$. If the cardinality of the set of matchers is greater than the specified *bound*, the matchers beyond that bound are omitted. The *module* can be any module or theory declared in the current Maude session; as usual, if the **in** clause is omitted, the current module is used.

For more details, see Section 14.13.

get {***irredundant***} **variants** {[***bound***]} {**in** ***module*** **:**} ***term*** {**such that** ***term"***$_1$**,** ..., ***term"***$_n$ ***irreducible***} **.**

Compute incrementally a set of most general variants of the given term in the (supported) equational theory of the given module, where the equations of interest must be declared with the `variant` attribute. The keyword `irredundant` is useful for theories that do have the finite variant property, since it will provide the set of most general variants of the given term. If the cardinality of the set of variants is greater than the specified *bound*, the variants beyond that bound are omitted. It may be necessary to satisfy an optional irreducibility condition on extra terms *term"*$_1$, ..., *term"*$_n$. The *module* can be any module or theory declared in the current Maude session; as usual, if the **in** clause is omitted, the current module is used.

For examples and more details, see Section 14.4.

{**f**}**vu-narrow** {[***bound*** {**,*depth***}]} {**in** ***module*** **:**} ***pattern1 searchtype pattern2*** {**such that** ***term"***$_1$**,** ..., ***term"***$_n$ ***irreducible***} **.**

Performs a breadth-first narrowing search using variant-based unification starting at an initial state *pattern1* (a term with variables) to a final state (a term with variables, possibly shared with the initial term) that unifies *pattern2* and satisfies an optional irreducibility condition on extra terms *term"*$_1$, ..., *term"*$_n$. If the letter `f` is added at the beginning, a *folding* narrowing search is used instead. Possible values for *searchtype* are (but their meaning differ from that of the `search command` of Section 5.4.3)

`=>1`	one step proof
`=>+`	one or more steps proof
`=>*`	zero or more steps proof
`=>!`	only canonical final states, that cannot be further narrowed, are allowed as solutions

The optional *bound* argument provides an upper bound in the number of solutions to be found; if it is omitted, infinity is assumed.

The optional *depth* argument indicates the maximum depth of the search. If it is omitted, infinity is assumed. It is also possible to give a depth bound without giving a bound on the number of solutions returned by requesting a search of the form `vu-narrow [,m] ...`.

The search type `=>1` is an abbreviation of the search type `=>+` with the depth bound set to `1`.

As usual, if the **in** clause is omitted, the current module is assumed.

For examples, see Section 15.6.

23.7 SMT commands

check {**in** ***module*** **:**} ***term*** **.**

Cause the connected SMT solver to be queried with the specified term, containing only constants and operator from the `smt.maude` signatures. If the **in** clause is omitted, the current module is assumed.

For examples and more details, see Section 16.

23.8 Tracing commands

Tracing produces detailed information about each rewrite performed and each conditional rewrite attempted. Since this typically results in an unmanageably huge volume of output, there are commands to control what is actually displayed.

set trace on . / set trace off .
These commands turn tracing on and off. If tracing is turned on, all trace information will be generated internally, even if none of it is displayed, thus considerably slowing the speed of interpretation.

set trace condition on . / set trace condition off .
Determines whether the evaluations of conditions are traced.

set trace whole on . / set trace whole off .
Determines whether the whole term is printed before and after a rewrite. Regarding strategy definitions, it determines whether the subject term is printed in each trace.

set trace substitution on . / set trace substitution off .
Determines whether the substitution is printed.

set trace mb on . / set trace mb off .
Determines whether membership axiom applications are printed.

set trace eq on . / set trace eq off .
Determines whether equation applications are printed.

set trace rl on . / set trace rl off .
Determines whether rule applications are printed.

set trace sd on . / set trace sd off .
Determines whether strategy definition applications are printed.

set trace select on . / set trace select off .
Determines whether only trace information for selected operator symbols is printed (rather than all symbols).

trace select *symbols* . / trace deselect *symbols* .
Selects/deselects operator symbols and labels from the current module for tracing with the select option. Examples:

```
trace select foo bar baz .
trace deselect baz .
```

trace exclude *modules* . / trace include *modules* .
Controls which modules are traced. Examples:

```
trace exclude META-LEVEL .
trace include MY-MOD1 MY-MOD2 .
```

set trace rewrite on . / set trace rewrite off .
Determines whether the redex and its replacement are printed.

set trace body on . / set trace body off .
Determines whether the "start of rewrite" line (i.e., the one beginning with *'s) and the body of the equation/rule/membership/strategy defintion being used are printed; if turned off, just the label and the substitution are printed. By setting both body and rewrite to **off** (see previous command), these options reduce a trace to a list of labels much like that produced by the `show path labels` *`number`* command.

set trace builtin on . / set trace builtin off .
Determines whether trace information for built-in operator symbols is printed.

23.9 Print attribute commands

In print attribute mode, when a statement is executed, the items in its `print` attribute are printed, with variables taking their value in the current substitution.

set print attribute on . / set print attribute off .
These commands turn print attribute mode on and off. It is **off** by default.

set print attribute newline on . / set print attribute newline off .
These commands determine whether a newline is printed following the items of a print attribute. By default a newline is printed (even if there are no items).

Note that print attribute mode is like trace mode, break mode, and profile mode in that in this mode all execution takes the slow path. This is true even if no print attributes are encountered.

23.10 Print option commands

set print mixfix on . / set print mixfix off .
Controls whether operators with mixfix syntax are printed in either mixfix or prefix form. User-defined syntax is supported for pretty-printing, even though it is not currently supported for parsing. It is sometimes advantageous to have uniform prefix notation for output; for example, if the output is going to be postprocessed by some other tool. Default is **on**.

set print graph on . / set print graph off .
If **on**, terms that are internally represented by graphs (currently, result terms together with terms being reduced and terms in substitutions during tracing) are printed as graph representations rather than as terms, together with the number of operator symbols in the full term. This can be useful in some pathological cases where the size of the term is exponential on the size of the graph. Default is **off**.

set print flattened on . / set print flattened off .
Controls whether arguments under operators with the associative attribute are printed in flattened form or not. Default is **on**.

set print with parentheses on . / set print with parentheses off .
If on, mixfix terms are printed with additional parentheses to make grouping explicit. Default is **off**.

set print with aliases on . / set print with aliases off .
Controls if variables aliases are used. Default is **on**.

set print number on . / set print number off .
Controls if special output convention for natural numbers is used. Default is **on**.

set print rational on . / set print rational off .
Controls if special output convention for rational numbers is used. Default is **on**.

set print color on . / set print color off .
Controls if reduction status coloring is used. Default is **off**.

set print format on . / set print format off .
Controls if format attributes are obeyed. Default is **on**.

set print conceal on . / set print conceal off .
Controls if argument hiding is used. Default is **off**.

print conceal *symbols* . / print reveal *symbols* .
Controls which operators have their arguments hidden.

set print constants with sorts on . / set print constants with sorts off .
Controls whether constants c of sort s are printed as `(`c`) .`s even if desambiguation is not strictly needed. Default is **off**.

23.11 Show option commands

set show stats on . / set show stats off .
Determines whether the number of rewrites is printed with the results of the `reduce`, `rewrite`, and `continue` commands in Section 23.2. Default is **on**.

set show loop stats on . / set show loop stats off .
As above but for loop mode.

set show timing on . / set show timing off .
Determines whether the cpu and real time used during rewriting is printed with the results of the `reduce`, `rewrite`, and `continue` commands in Section 23.2. Default is **on**.

set show loop timing on . / set show loop timing off .
As above but for loop mode.

set show command on . / set show command off .
Determines whether the full form of certain commands is printed before they are executed. Default is **on**.

set show breakdown on . / set show breakdown off .
Determines whether a breakdown of rewrites is dispayed. Default is **off**.

set show gc on . / set show gc off .
Determines which message is printed when a garbage collect is performed. Default is **off**.

set show advisories on . / set show advisories off .
Determines whether advisories are displayed. Default is **on**.

23.12 Show commands

show modules .
Lists the names of all the modules that are currently in the module database maintained by the system.

show module {*module*} .
Prints out a representation of the given module (or of the current module if none is given).

show all {*module*} .
Prints out a *flattened* representation of the given module (or of the current module if none is given).

show sorts {*module*} .
Prints out a representation of the sort and subsort information for the given module (or for the current module if none is given).

show ops {*module*} .
Lists the operators in the given module (or in the current module if none is given).

show vars {*module*} .
Lists the variables in the given module (or in the current module if none is given).

show mbs {*module*} .
Lists the membership axioms in the given module (or in the current module if none is given).

show eqs {*module*} .
Lists the equations in the given module (or in the current module if none is given).

show rls {*module*} .
Lists the rules in the given module (or in the current module if none is given).

show strats {*module*} .
Lists the strategies in the given module (or in the current module if none is given).

show sds {*module*} .
Lists the strategy definitions in the given module (or in the current module if none is given).

show components {*module*} .
Lists the connected components (kinds) of the poset of sorts for the given module (or for the current module if none is given).

show summary {*module*} .
Shows a summary of statistics for the context free grammar and term rewriting system generated for the given module (or for the current module if none is given).

show views .
Lists the names of all the views that are currently in the view database maintained by the system.

show view {***view***} **.**
Prints out the given view (or of the last view entered into the system if none is given).

23.13 Profiler commands

set profile on . / set profile off .
Turns profiling on and off. Default is **off**.

set clear profile on . / set clear profile off .
Controls whether profile is clear before each command. Default is **on**.

show profile {***module***} **.**
Shows current profile for the given module (or in the current module if none is given). It shows both percentages and absolute rewrite counts.

23.14 Debugger commands

set break on . / set break off .
Controls whether break points are obeyed.

break select *symbols* . / break deselect *symbols* .
Selects/deselects operator symbols and labels from the current module for break points with the select option. Examples:

```
break select foo bar baz .
break deselect baz .
```

resume .
Only usable from the debugger. Exits the debugger and resumes the current rewriting activity.

abort .
Only usable from the debugger. Exits the debugger and abandons the current rewriting activity.

step .
Only usable from the debugger. Performs a single step of the current rewriting activity with tracing switched on.

where .
Only usable from the debugger. Prints the stack of pending rewrite tasks together with explanations of how they arose.

Moreover, any command that involves the execution of rewriting or narrowing steps can be prefixed by the `debug` word to drop into the debugger before the first step takes place. These commands are `reduce`, `rewrite`, `frewrite`, `erewrite`, `continue` (in Section 23.2), `search` (in

Section 23.4), `srewrite`, `dsrewrite` (in Section 23.5), `variant unify`, `variant match`, `get variants`, `vu-narrow` and `fvu-narrow` (in Section 23.6).

23.15 Miscellaneous commands

parse {**in** ***module*** **:**} ***term*** **.**
Causes the specified term to be parsed using the signature of the given module. If the **in** clause is omitted, the current module is assumed.

select ***module*** **.**
Selects a named module to be the current module. All commands that require a module refer to the current module, unless a module is explicitly given. The current module is usually the last module entered or used; for example, after the command `show module AMODULE`, the `AMODULE` module becomes the current module.

set protect ***module*** **on . / set protect** ***module*** **off .**
Adds or removes the named module from the set of modules that are automatically imported in `protecting` mode in every module.

set extend ***module*** **on . / set extend** ***module*** **off .**
Adds or removes the named module from the set of modules that are automatically imported in `extending` mode in every module.

set include ***module*** **on . / set include** ***module*** **off .**
Adds or removes the named module from the set of modules that are automatically imported in `including` mode in every module.

set verbose on . / set verbose off .
Controls display of extra information, depending on command. Default is **off**.

set clear memo on . / set clear memo off .
Controls whether the memoization tables are cleared before each command.

do clear memo { ***module*** } **.**
Clear the memoization tables of the given module (or of the current module if none is given).

23.16 System level commands

These commands control system level activities. Unlike all the above commands they are not followed by a period.

pwd
Prints the path of the working directory.

ls {***flags***} {***directories***}
Runs the UNIX `ls` command to list the files in the specified directories or working directory if none specified. The allowable flags depend on your local implementation of `ls`. Example:

```
ls -lF /usr/bin/usr/local
```

ll Runs the common UNIX abbreviation `ll` (for `ls -l`).

cd *directory-name*
Changes the working directory to *directory-name*.

pushd *directory-name*
Saves the current working directory on a stack and then changes the working directory to *directory-name*.

popd
Changes the working directory to that which is on the top of the directory stack and pops the directory stack.

in *file-name*
Causes a specified file to be included at this point. For files specified by a bare file name, it checks (with `.maude`, `.fm`, `.obj` extensions) if the filename is in one of these locations: (a) the current directory; (b) the directories in the `MAUDE_LIB` environment variable, and (c) the directory containing the executable. Otherwise, the full file name must be given, together with a full path name if the file is not in the current working directory. The `in` command may be nested, i.e., the included file may contain `in` commands. Example:

```
in ../Examples/foo.maude
```

Notice that compilation of operator declarations and statements is done lazily, so that the module is not necessarily fully compiled when included. This implies that some warnings and advisories will only show up when a reduction actually takes place in the module. This also holds for a module that is entered by writing it in the prompt instead of a file.

load *file-name*
Performs the same job as `in` but does not produce detailed output as modules are entered. Example:

```
load ../Examples/foo.maude
```

sload *file-name*
Performs the same job as `load` but loads the named file only if it has changed (determined by the file system's modify time) since it was last read (via `in`, `load`, `sload` or command line argument).[1] Example:

```
sload ../Examples/foo.maude
```

eof Causes the interpreter to respond as if it had reached the end of file.

quit
Causes the interpreter to exit.

[1]Note that since modify times are stored as seconds, there is a potential race condition if a file is modified, loaded, and modified again within the space of a second, so `sload` is best used to load relatively static files rather than machine generated/updated ones.

Chapter 24

Core Maude Grammar

This chapter describes the syntax of Maude using the following extended BNF notation: the symbols '(' and ')' are used as metaparentheses; the symbol '|' is used to separate alternatives; square bracket pairs, '[' and ']', enclose optional syntax; '*' indicates zero or more repetitions of preceding unit; '+' indicates one or more repetitions of preceding unit; and the string "`x`" denotes `x` literally. As an application of this notation, `A( , A)*` indicates a non-empty list of `A`'s separated by commas. Finally, `%%%` indicates comments in the syntactic description, as opposed to comments in the Maude code.

24.1 The grammar

```
⟨MaudeTop⟩ ::=
    ( ⟨SystemCommand⟩ | ⟨Command⟩ | ⟨DebuggerCommand⟩ |
      ⟨Module⟩ | ⟨Theory⟩ | ⟨View⟩ )+

⟨SystemCommand⟩ ::= in ⟨FileName⟩ | load ⟨FileName⟩ | sload ⟨FileName⟩ |
    quit | eof | popd | pwd |
    cd ⟨Directory⟩ | push ⟨Directory⟩ |
    ls [ ⟨LsFlag⟩ ] [ ⟨Directory⟩ ]

⟨Command⟩ ::= select ⟨ModId⟩ . |
    parse [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    [ debug ] reduce [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    [ debug ] rewrite [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    [ debug ] frewrite [ ⟨DoubleBound⟩ ] [ in ⟨ModId⟩ :
        ] ⟨Term⟩ . |
    [ debug ] erewrite [ ⟨DoubleBound⟩ ] [ in ⟨ModId⟩ : ]
        ⟨Term⟩ . |
    [ debug ] ( srewrite | dsrewrite ) [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ]
        ⟨Term⟩ using ⟨Strat⟩ . |
    check [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    ( match | xmatch ) [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ]
        ⟨Term⟩ <=? ⟨Term⟩ [ such that ⟨Condition⟩ ] . |
    [ debug ] variant match [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ]
        ⟨Term⟩ <=? ⟨Term⟩ [ such that ⟨Condition⟩ ] . |
```

```
    [ irredudant ] unify [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ]
        ⟨UnificationEquation⟩ ( /\ ⟨UnificationEquation⟩ )* . |
    [ debug ] [ filtered ] variant unify [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ]
        ⟨UnificationEquation⟩ ( /\ ⟨UnificationEquation⟩ )* . |
    [ debug ] get [ irredundant ] variants [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
   [ debug ] ( search | vu-narrow ) [ ⟨DoubleBound⟩ ] [ in ⟨ModId⟩ : ]
        ⟨Term⟩ ⟨SearchType⟩ ⟨Term⟩ [ such that ⟨Condition⟩ ] . |
    [ debug ] continue ⟨Nat⟩ . |
    loop [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    ( ⟨TokenString⟩ ) |
    trace ( select | deselect | include | exclude )
        ( ⟨OpId⟩ | ( ⟨OpForm⟩ ) )+ . |
    print ( conceal | reveal ) ( ⟨OpId⟩ | ( ⟨OpForm⟩ ) )+ . |
    break ( select | deselect ) ( ⟨OpId⟩ | ( ⟨OpForm⟩ ) )+ . |
    show ⟨ShowItem⟩ [ ⟨ModId⟩ ] . |
    show view [ ⟨ViewId⟩ ] . |
    show modules . |
    show views . |
    show search graph . |
    show path [ labels ] ⟨Nat⟩ .
    do clear memo [ ⟨ModId⟩ ] . |
    set ⟨SetOption⟩ ( on | off ) .

⟨DoubleBound⟩ ::= [ ⟨Nat⟩ [ , ⟨Nat⟩ ] ] | [ , ⟨Nat⟩ ]

⟨SearchType⟩ ::= =>1 | =>+ | =>* | =>!

⟨UnificationEquation⟩ ::= ⟨Term⟩ =? ⟨Term⟩

⟨ShowItem⟩ ::= module | all | sorts | ops | vars | mbs |
    eqs | rls | strats | sds | summary | kinds | profile

⟨SetOption⟩ ::= show ⟨ShowOption⟩ |
    print ⟨PrintOption⟩ |
    trace [ ⟨TraceOption⟩ ] |
    break | verbose | profile |
    clear ( memo | rules | profile ) |
    protect ⟨ModId⟩ |
    extend ⟨ModId⟩ |
    include ⟨ModId⟩

⟨ShowOption⟩ ::= advise | stats | loop stats | timing |
    loop timing | breakdown | command | gc

⟨PrintOption⟩ ::= mixfix | flat | with parentheses |
    with aliases | conceal | number | rat | color |
    format | graph | attribute | attribute newline |
    constants with sorts
```

```
⟨TraceOption⟩ ::= condition | whole | substitution | select |
    mbs | eqs | rls | sds | rewrite | body

⟨DebuggerCommand⟩ ::= resume . | abort . | step . | where .

⟨Module⟩ ::= fmod ⟨ModId⟩ [ ⟨ParameterList⟩ ] is ⟨ModElt⟩* endfm |
    mod ⟨ModId⟩ [ ⟨ParameterList⟩ ] is ⟨ModElt'⟩* endm |
    smod ⟨ModId⟩ [ ⟨ParameterList⟩ ] is ⟨SmodElt⟩* endsm

⟨Theory⟩ ::= fth ⟨ModId⟩ is ⟨ModElt⟩* endfth |
    th ⟨ModId⟩ is ⟨ModElt'⟩* endth |
    sth ⟨ModId⟩ is ⟨SmodElt⟩* endsth

⟨View⟩ ::= view ⟨ViewId⟩ [ ⟨ParameterList⟩ ] from ⟨ModExp⟩ to ⟨ModExp⟩ is
             ⟨ViewElt⟩*
           endv

⟨ParameterList⟩ ::= { ⟨ParameterDecl⟩ ( , ⟨ParameterDecl⟩ )* }

⟨ParameterDecl⟩ ::= ⟨ParameterId⟩ :: ⟨ModExp⟩

⟨ModElt⟩ ::= including ⟨ModExp⟩ . |
    extending ⟨ModExp⟩ . |
    protecting ⟨ModExp⟩ . |
    sorts ⟨Sort⟩+ . |
    subsorts ⟨Sort⟩+ ( < ⟨Sort⟩+ )+ . |
    op ⟨OpForm⟩ : ⟨Type⟩* ⟨Arrow⟩ ⟨Type⟩ [ ⟨Attr⟩ ] . |
    ops ( ⟨OpId⟩ | ( ⟨OpForm⟩ ) )+ : ⟨Type⟩* ⟨Arrow⟩ ⟨Type⟩
          [ ⟨Attr⟩ ] . |
    vars ⟨VarId⟩+ : ⟨Type⟩ . |
    ⟨Statement⟩ [ ⟨StatementAttr⟩ ] .

⟨ViewElt⟩ ::= var ⟨varId⟩+ : ⟨Type⟩ . |
    sort ⟨Sort⟩ to ⟨Sort⟩ . |
    label ⟨LabelId⟩ to ⟨LabelId⟩ . |
    op ⟨OpForm⟩ to ⟨OpForm⟩ . |
    op ⟨OpForm⟩ : ⟨Type⟩* ⟨Arrow⟩ ⟨Type⟩ to ⟨OpForm⟩ . |
    op ⟨Term⟩ to term ⟨Term⟩ . |
    strat ⟨StratId⟩ to ⟨StratId⟩ . |
    strat ⟨StratId⟩ [ : ⟨Type⟩* ] @ ⟨Type⟩ to ⟨StratId⟩ . |
    strat ⟨StratCall⟩ to expr ⟨Strat⟩ .

⟨ModExp⟩ ::= ⟨ModId⟩ |
    ( ⟨ModExp⟩ ) |
    ⟨ModExp⟩ + ⟨ModExp⟩ |
    ⟨ModExp⟩ * ⟨Renaming⟩
    ⟨ModExp⟩ { ⟨ViewId⟩ ( , ⟨ViewId⟩ )* }

⟨Renaming⟩ ::= ( ⟨RenamingItem⟩ ( , ⟨RenamingItem⟩ )* )
```

```
⟨RenamingItem⟩ ::= sort ⟨Sort⟩ to ⟨Sort⟩ |
    label ⟨LabelId⟩ to ⟨LabelId⟩ |
    op ⟨OpForm⟩ ⟨ToPartRenamingItem⟩ |
    op ⟨OpForm⟩ : ⟨Type⟩* ⟨Arrow⟩ ⟨Type⟩ ⟨ToPartRenamingItem⟩
    strat ⟨StratId⟩ to ⟨StratId⟩ |
    strat ⟨StratId⟩ [ : ⟨Type⟩* ] @ ⟨Type⟩ to ⟨StratId⟩ |

⟨ToPartRenamingItem⟩ ::= to ⟨OpForm⟩ [ ⟨Attr⟩ ]

⟨Arrow⟩ ::= -> | ~>

⟨Type⟩ ::= ⟨Sort⟩ | ⟨Kind⟩

⟨Kind⟩ ::= [ ⟨Sort⟩ ( , ⟨Sort⟩ )* ]

⟨Sort⟩ ::= ⟨SortId⟩ | ⟨Sort⟩ { ⟨Sort⟩ ( , ⟨Sort⟩ )* }

⟨ModElt'⟩ ::= ⟨ModElt⟩ |
    ⟨Statement'⟩ [ ⟨StatementAttr⟩ ] .

⟨SmodElt⟩ ::= including ⟨ModExp⟩ . |
    extending ⟨ModExp⟩ . |
    protecting ⟨ModExp⟩ . |
    vars ⟨VarId⟩+ : ⟨Type⟩ . |
    strats ⟨StratId⟩+ [ : ⟨Type⟩* ] @ ⟨Type⟩ [ ⟨StratAttr⟩ ] .
    ⟨StratStatement⟩ [ ⟨StatementAttr⟩ ] .

⟨Statement⟩  ::=  mb [ ⟨Label⟩ ] ⟨Term⟩ : ⟨Sort⟩ |
    cmb [ ⟨Label⟩ ] ⟨Term⟩ : ⟨Sort⟩ if ⟨Condition⟩ |
    eq [ ⟨Label⟩ ] ⟨Term⟩ = ⟨Term⟩ |
    ceq [ ⟨Label⟩ ] ⟨Term⟩ = ⟨Term⟩ if ⟨Condition⟩

⟨Statement'⟩ ::= rl [ ⟨Label⟩ ] ⟨Term⟩ => ⟨Term⟩ |
    crl [ ⟨Label⟩ ] ⟨Term⟩ => ⟨Term⟩ if ⟨Condition'⟩

⟨StratStatement⟩ ::= sd ⟨StratCall⟩ := ⟨Strat⟩ |
    csd ⟨StratCall⟩ := ⟨Strat⟩ if ⟨Condition⟩

⟨Label⟩ ::= [ ⟨LabelId⟩ ] :

⟨Condition⟩ ::= ⟨ConditionFragment⟩ ( /\ ⟨ConditionFragment⟩ )*

⟨Condition'⟩ ::= ⟨ConditionFragment'⟩
                 ( /\ ⟨ConditionFragment'⟩ )*

⟨ConditionFragment⟩ ::= ⟨Term⟩ = ⟨Term⟩ | ⟨Term⟩ := ⟨Term⟩
                        | ⟨Term⟩ : ⟨Sort⟩
```

```
⟨ ConditionFragment' ⟩ ::= ⟨ ConditionFragment ⟩ | ⟨ Term ⟩ => ⟨ Term ⟩

⟨ Attr ⟩ ::=
    [ ( assoc | comm |
        [ left | right ] id: ⟨ Term ⟩ |
        idem | iter | memo | ditto |
        config | obj | msg | ctor |
        metadata ⟨ StringId ⟩
        strat ( ⟨ Nat ⟩+ ) |
        poly ( ⟨ Nat ⟩+ ) |
        frozen [ ( ⟨ Nat ⟩+ ) ] |
        prec ⟨ Nat ⟩ |
        gather ( ( e | E | & )+ ) |
        format ( ⟨ Token ⟩+ ) |
        special ( ⟨ Hook ⟩+ ) )+ ]

⟨ StratAttr ⟩ ::= [ metadata ⟨ StringId ⟩ ]

⟨ StatementAttr ⟩ ::=
    [ ( nonexec |
        metadata ⟨ StringId ⟩ |
        label ⟨ LabelId ⟩ |
        print ⟨ PrintItem ⟩* )+ ]

⟨ StatementAttrEq ⟩ ::=
    [ ( ⟨ StatementAttr ⟩ | otherwise | variant )+ ]

⟨ StatementAttrRl ⟩ ::=
    [ ( ⟨ StatementAttr ⟩ | narrowing )+ ]

⟨ PrintItem ⟩ ::= ⟨ StringId ⟩ | ⟨ VarId ⟩ | ⟨ VarAndSortId ⟩

⟨ Hook ⟩ ::= id-hook ⟨ Token ⟩ [ ( ⟨ TokenString ⟩ ) ] |
    ( op-hook | term-hook ) ⟨ Token ⟩ ( ⟨ TokenString ⟩ )

⟨ Strat ⟩ ::= idle | fail |
    ⟨ RuleApp ⟩ | top( ⟨ RuleApp ⟩ ) |
    ⟨ Strat ⟩ ? ⟨ Strat ⟩ : ⟨ Strat ⟩ |
    ⟨ TestVariant ⟩ ⟨ Term ⟩ [ such that ⟨ Condition ⟩ ]
    ⟨ Strat ⟩ ; ⟨ Strat ⟩ |
    ⟨ Strat ⟩ | ⟨ Strat ⟩ |
    ⟨ Strat ⟩ * |
    ⟨ MrewVariant ⟩ ⟨ Term ⟩ [ such that ⟨ Condition ⟩ ] by ⟨ VarStratList ⟩
    ⟨ StratCall ⟩
    ⟨ Strat ⟩ +
    ⟨ Strat ⟩ or-else ⟨ Strat ⟩
    not( ⟨ Strat ⟩ )
    ⟨ Strat ⟩ !
    try( ⟨ Strat ⟩ )
```

```
    test( ⟨Strat⟩ )

⟨RuleApp⟩ ::= ⟨LabelId⟩ [ [ ⟨Substitution⟩ ] ] [ { ⟨Strat⟩ (, ⟨Strat⟩)* } ]

⟨Substitution⟩ ::= ⟨VarId⟩ <- ⟨Term⟩ |
    ⟨Substitution⟩ , ⟨Substitution⟩

⟨StratCall⟩ ::= ⟨StratId⟩ [ () ] |
    ⟨StratId⟩ ( ⟨Term⟩ (, ⟨Term⟩)* )

⟨VarStratList⟩ ::= ⟨VarId⟩ using ⟨Strat⟩ |
    ⟨VarStratList⟩ , ⟨VarStratList⟩

⟨TestVariant⟩ ::= match | xmatch | amatch

⟨MrewVariant⟩ ::= matchrew | xmatchrew | amatchrew

⟨FileName⟩    %%% OS dependent
⟨Directory⟩   %%% OS dependent
⟨LsFlag⟩      %%% OS dependent

⟨StringId⟩    %%% characters enclosed in double quotes "..."
⟨ModId⟩       %%% simple identifier, by convention all capitals
⟨ViewId⟩      %%% simple identifier, by convention capitalized
⟨ParameterId⟩ %%% simple identifier, by convention single capital
⟨SortId⟩      %%% simple identifier, by convention capitalized
⟨VarId⟩       %%% simple identifier, by convention capitalized
⟨VarAndSortId⟩ %%% an identifier consisting of a variable name
                   followed by a colon followed by a sort name
⟨OpId⟩        %%% identifier possibly with underscores
⟨OpForm⟩ ::= ⟨OpId⟩ | ( ⟨OpForm⟩ ) | ⟨OpForm⟩+
⟨Nat⟩         %%% a natural number
⟨Term⟩ ::= ⟨Token⟩ | ( ⟨Term⟩ ) | ⟨Term⟩+
⟨Token⟩       %%% sequence of printable ASCII characters delimited by
                  whitespace. The symbols (, ), [, ], {, } and comma form
                  separate tokens themselves, unless backquoted
⟨TokenString⟩ ::= ⟨Token⟩ | ( ⟨TokenString⟩ ) | ⟨TokenString⟩*
⟨LabelId⟩     %%% simple identifier
⟨StratId⟩     %%% simple identifier
```

In parsing module expressions, instantiation has higher precedence than renaming, which in turn has higher precedence than summation.

24.2 Synonyms

```
sort = sorts
subsort = subsorts
var = vars
```

Command only synonyms:

```
advise = advisory = advisories
alias = aliases
attr = attribute
cmd = command
cond = condition
cont = continue
dsrew = dsrewrite
eqs = eq
erew = erewrite
flat = flattened
frew = frewrite
irred = irreducible
kinds = components
label = labels
mbs = mb
nar = narrow
paren = parens = parentheses
q = quit
rat = rational
red = reduce
rew = rewrite
rls  = rl = rule = rules
s.t. = such that
srew = srewrite
subst = substitution
sds = sd
```

Module only synonyms:

```
assoc = associative
ceq = cq
comm = commutative
config = configuration
ctor = constructor
ex = extending
id: = identity:
idem = idempotent
inc = including
iter = iterated
msg = message
obj = object
owise = otherwise
poly = polymorphic
prec = precedence
pr = protecting
strat = strategy       %%% reduction strategy attribute
strats = strat         %%% rewriting strategy declaration
```

24.3 Lexical Issues

Tokens are sequences of printable ASCII characters delimited by white space, except that '`(`', '`)`', '`[`', '`]`', '`{`', '`}`', and '`,`' are always considered as single character tokens, unless backquoted.

Single line comments are started by one of `***` or `---`, and ended by the end of line. Multiline comments are started by `***(` and ended by `)`. Parentheses (whether backquoted or not) must balance within multiline comments.

String identifiers use C backslash conventions [87, Section A2.5.2].

Bibliography

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986. Cited in 8.1.

[2] Gul Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In A. Cerone and H. Wiklicky, editors, *Proceedings Third Workshop on Quantitative Aspects of Programming Languages, QAPL'05, Edinburgh, UK, April 2005*, volume 153(2) of *Electronic Notes in Theoretical Computer Science*, pages 213–239. Elsevier, 2006. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 1.4.

[3] María Alpuente, Santiago Escobar, and José Iborra. Modular termination of basic narrowing. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008. Cited in 13.5.1.

[4] María Alpuente, Moreno Falaschi, and Germán Vidal. Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998. Cited in 15.1.

[5] María Alpuente, José Iborra, and Santiago Escobar. Termination of narrowing revisited. *Theoretical Computer Science*, 410(46):4608–4625, 2009. Cited in 13.5.1, 14.8, 5.

[6] Krzysztof R. Apt. Chapter 10 - Logic programming. In Jan Van Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 493 – 574. Elsevier, Amsterdam, 1990. Cited in 15.6.

[7] Thomas Arts and Hans Zantema. Termination of logic programs using semantic unification. In Maurizio Proietti, editor, *Logic Programming Synthesis and Transformation, 5th International Workshop, LOPSTR'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings*, volume 1048 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 1996. Cited in 15.1.

[8] Kyungmin Bae, Santiago Escobar, and José Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. Cited in 13.5.2, 15.2, 15.7.

[9] David Basin, Manuel Clavel, and José Meseguer. Reflective metalogical frameworks. *ACM Transactions on Computational Logic*, 5(3):528–576, 2004. Cited in 1.4.

[10] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 1980. Cited in 1.1.2, 7.1.

[11] Hans-Juergen Boehm, Russell R. Atkinson, and Michael F. Plass. Ropes: An alternative to strings. *Softw. Pract. Exp.*, 25(12):1315–1330, 1995. Cited in 7.8.

[12] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002. Cited in 1.7, 3.

[13] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *Int. J. Found. Comput. Sci.*, 12(1):69–95, 2001. Cited in 10.

[14] Christopher Bouchard, Kimberly A. Gero, Christopher Lynch, and Paliath Narendran. On forward closure and the finite variant property. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, volume 8152 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2013. Cited in 14.2.

[15] Alexandre Boudet. Unification in a combination of equational theories: an efficient algorithm. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, volume 449 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 1990. Cited in 13.7.1.

[16] Alexandre Boudet, Evelyne Contejean, and Hervé Devie. A new AC-unification algorithm with an algorithm for solving systems of diophantine equations. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS 1990)*, pages 289–299. IEEE Computer Society Press, June 1990. Cited in 13.7, 13.7.1, 13.7.3.

[17] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000. Cited in 1.2, 3.5, 4, 4.3, 4.6, 4.7, 6.1.1, 13.5.1.

[18] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. Cited in 10.

[19] Roberto Bruni and José Meseguer. Generalized rewrite theories. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003. Cited in 1.2, 5.3, 6.1.1.

[20] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. Cited in 13.7.3.

[21] Rod Burstall and Joseph A. Goguen. The semantics of Clear, a specification language. In Dines Bjørner, editor, *Abstract Software Specifications, 1979 Copenhagen Winter School, January 22 - February 2, 1979, Proceedings*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer, 1980. Cited in 6, 6.3, 21.3.1.

[22] Fabricio Chalub and Christiano Braga. Maude MSOS tool. In Grit Denker and Carolyn Talcott, editors, *Proceedings Sixth International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1–2, 2006*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 3–17. Elsevier, 2007. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 1.5.

[23] Andrew Cholewa, José Meseguer, and Santiago Escobar. Variants of variants and the finite variant property. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/47117, 2014. Cited in 13.5.1, 14.2.

[24] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. Cited in 12, 12.3, 12.3.

[25] Manuel Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000. Cited in 17, 17.7.

[26] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. A tutorial on Maude. SRI International, March 2000, `http://maude.cs.illinois.edu/maude1/tutorial/`. Cited in 17.7.

[27] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Towards Maude 2.0. In Futatsugi [74], pages 294–315. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 9.3.1.

[28] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002. Cited in 17.1, 17.7.

[29] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. Cited in 1.7, 1, 5.3, 9.3.2, 11.4, 12.5.

[30] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational proving tools by reflection in rewriting logic. In Kokichi Futatsugi, Ataru T. Nakagawa, and Tetsuo Tamai, editors, *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000. `http://maude.cs.illinois.edu/papers/`. Cited in 1.3, 1.4, 4.7, 6.1.1, 6.3.2.

[31] Manuel Clavel, Francisco Durán, Steven Eker, José Meseguer, and Mark-Oliver Stehr. Maude as a formal meta-tool. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20–24, 1999 Proceedings, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer, 1999. Cited in 1.4.

[32] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In Meseguer [105], pages 126–148. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 17.

[33] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002. Cited in 1.1.2, 17.

[34] Manuel Clavel, José Meseguer, and Miguel Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. In Gadducci and Montanari [76], pages 91–107. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 1.1.2, 17.

[35] Alain Colmerauer, Henry Kanoui, and Michel van Caneghem. Étude et réalisation d'un système Prolog. techreport, Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II, 1979. Cited in 10.5.

[36] Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: how to get rid of some algebraic properties. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005. Cited in 13.5.1, 14.2, 14.2, 14.8, 15.2.

[37] Evelyn Contejean and Hervé Devie. An efficient incremental algorithm for solving systems of linear diophantine equations. *Information and Computation*, 113(1):143–172, 1994. Cited in 7.15, 7.15, 13.7.

[38] Evelyne Contejean, Claude Marché, and Xavier Urbain. CiME, 2004. Available at `http://cime.lri.fr/`. Cited in 13.6, 13.7.

[39] Nachum Dershowitz. Goal solving as operational semantics. In *International Logic Programming Symposium, Portland, OR*, pages 3–17. MIT Press, December 1995. Cited in 15.1.

[40] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. North-Holland, 1990. Cited in 4.8.

[41] Arie van Deursen. *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994. Cited in 18.3.

[42] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996. Cited in 3, 10, 18.3.

[43] Eric Domenjoud. Solving systems of linear diophantine equations: An algebraic approach. In Andrzej Tarlecki, editor, *Mathematical Foundations of Computer Science 1991, 16th International Symposium, MFCS'91, Kazimierz Dolny, Poland, September 9-13, 1991, Proceedings*, volume 520 of *Lecture Notes in Computer Science*, pages 141–150. Springer, 1991. Cited in 7.15.

[44] Francisco Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, Spain, 1999. `http://maude.cs.illinois.edu/papers/`. Cited in 6, 21, 22.8.

[45] Francisco Durán. The extensibility of Maude's module algebra. In Teodor Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2000. Cited in 21.

[46] Francisco Durán, Salvador Lucas, Claude Marché, José Meseguer, and Xavier Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21(1-2):59–88, 2008. Cited in 1.3, 4.7.

[47] Francisco Durán, Salvador Lucas, and José Meseguer. MTT: The Maude termination tool (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008. Cited in 1.3, 4.7, 21.5.

[48] Francisco Durán and José Meseguer. A Church-Rosser checker tool for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, 2000. `http://www.lcc.uma.es/~duran/CRC/`. Cited in 1.3.

[49] Francisco Durán and José Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner [89], pages 174–195. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 21.

[50] Francisco Durán and José Meseguer. The Maude specification of Full Maude. Technical report, Computer Science Laboratory, SRI International, February 1999. `http://maude.cs.illinois.edu/papers/`. Cited in 21.

[51] Francisco Durán and José Meseguer. Parameterized theories and views in Full Maude 2.0. In Futatsugi [74], pages 316–338. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 21.

[52] Francisco Durán and José Meseguer. Structured theories and institutions. *Theoretical Computer Science*, 309(1-3):357–380, 2003. Cited in 6.

[53] Francisco Durán and José Meseguer. Maude's module algebra. *Science of Computer Programming*, 66(2):125–153, 2007. Cited in 6.

[54] Francisco Durán and José Meseguer. A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In Peter Csaba Ölveczky, editor, *8th International Workshop on Rewriting Logic and its Applications, WLRA 2010, Paphos, Cyprus, March 20-21, 2010, Proceedings*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2010. Cited in 1.3, 4.7, 6.1.1, 13.5.3, 21.5.

[55] Francisco Durán and José Meseguer. A Maude coherence checker tool for conditional order-sorted rewrite theories. In Peter Csaba Ölveczky, editor, *8th International Workshop on Rewriting Logic and its Applications, WLRA 2010, Paphos, Cyprus, March 20-21, 2010, Proceedings*, volume 6381 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2010. Cited in 1.3, 13.5.3.

[56] Francisco Durán and José Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming*, 81(7-8):816–850, 2012. Cited in 1.3, 6.1.1, 13.5.3, 21.5.

[57] Steven Eker. Fast matching in combination of regular equational theories. In Meseguer [105], pages 90–109. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 1.1.3, 13.7.3.

[58] Steven Eker. Term rewriting with operator evaluation strategies. In Kirchner and Kirchner [89], pages 311–330. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 4.4.7, 4.4.7, 17.

[59] Steven Eker. Associative-commutative rewriting on large terms. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 14–29. Springer, 2003. Cited in 1.1.3.

[60] Steven Eker. Unification in Maude, January 2007. Talk at the "Protocol eXchange Seminar", Naval Postgraduate School. Available at `http://maude.cs.illinois.edu/talks/eker-unification.pdf`. Cited in 13.7.

[61] Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Deduction, strategies, and rewriting. In Myla Archer, Thierry Boy de la Tour, and César Muñoz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, August 16, 2006*, volume 174(11) of *Electronic Notes in Theoretical Computer Science*, pages 3–25. Elsevier, 2007. Cited in 10.

[62] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In Gadducci and Montanari [76], pages 143–168. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 12.3, 12.4.

[63] Santiago Escobar. Functional logic programming in maude. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, volume 8373 of *Lecture Notes in Computer Science*, pages 315–336. Springer, 2014. Cited in 15.8, 15.8.

[64] Santiago Escobar. Multi-paradigm programming in maude. In Vlad Rusu, editor, *Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Thessaloniki, Greece, June 14-15, 2018, Proceedings*, volume 11152 of *Lecture Notes in Computer Science*, pages 26–44. Springer, 2018. Cited in 15.8, 15.8.

[65] Santiago Escobar, Joe Hendrix, Catherine Meadows, and José Meseguer. Diffie-Hellman cryptographic reasoning in the Maude-NRL protocol analyzer. In Monica Nesi and Ralf Treinen, editors, *Proceedings Second International Workshop on Security and Rewriting Techniques, SecReT 2007, Paris, France, June 29, 2007*, 2007. Cited in 13.6.

[66] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006. Cited in 13.5.2.

[67] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2009. Cited in 13.5.2.

[68] Santiago Escobar and José Meseguer. Symbolic model checking of infinite-state systems using narrowing. In Franz Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2007. Cited in 13.5.2, 15.2, 15.7.

[69] Santiago Escobar, José Meseguer, and Ralf Sasse. Effectively checking the finite variant property. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*,

volume 5117 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2008. Cited in 13.5.1, 14.2.

[70] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and equational unification. In Grigore Roşu, editor, *Proceedings 7th International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29–30, 2008*, volume 238(3) of *Electronic Notes in Theoretical Computer Science*, pages 88–102. Elsevier, 2009. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 13.5.1.

[71] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. *Journal of Logic and Algebraic Programming*, 81(7-8):898–928, 2012. Cited in 13.5.1, 14.8, 15.2, 15.6, 15.7.

[72] François Fages. Associative-commutative unification. *Journal of Symbolic Computation*, 3(3):257–275, 1987. Cited in 13.7.1.

[73] M. Fay. First-order unification in an equational theory. In W. H. Joyner, editor, *Proceedings of the 4th Workshop on Automated Deduction, Austin, Texas, USA*, pages 161–167. Academic Press, 1979. Cited in 15.1.

[74] Kokichi Futatsugi, editor. *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 27, 51.

[75] Kokichi Futatsugi and Razvan Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998. Cited in 1.7.

[76] Fabio Gadducci and Ugo Montanari, editors. *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 34, 62, 136.

[77] Joseph Goguen and José Meseguer. Eqlog: Equality, types and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986. Cited in 15.1.

[78] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992. Cited in 3.8, 4.

[79] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph A. Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer Academic Publishers, 2000. Cited in 2, 1.1.2, 1.7, 3.9.1, 3.9.2, 4.4.7, 6, 6.3, 21.3.1.

[80] Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994. Cited in 15.1, 15.8.

[81] Joe Hendrix and José Meseguer. On the completeness of context-sensitive order-sorted specifications. Technical Report UIUCDCS-R-2007-2812, Computer Science Dept., University of Illinois at Urbana-Champaign, February 2007. Cited in 4.4.7, 4.7.

[82] Joe Hendrix and José Meseguer. Order-sorted equational unification revisited. In Günter Kniesel and Jorge Sousa Pinto, editors, *Proceedings Ninth International Workshop on Rule-Based Programming, RULE 2008, Hagenberg Castle, Austria*, volume 290 of *Electronic Notes in Theoretical Computer Science*, pages 37–50. Elsevier, 2012. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 13.6.

[83] Joe Hendrix, José Meseguer, and Hitoshi Ohsaki. A sufficient completeness checker for linear order-sorted specifications modulo axioms. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning - Third International Joint Conference, IJCAR 2006, Seattle, Washington, August 17 - 20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 151–155. Springer, 2006. Cited in 1.3, 4.7, 20.1.2.

[84] Jean-Marie Hullot. Canonical forms and unification. In Wolfgang Bibel and Robert A. Kowalski, editors, *Fifth Conference on Automated Deduction, CADE 1980, Les Arcs, France, July 8-11, 1980, Proceedings*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980. Cited in 13.5.1, 14.8, 15.1, 15.3, 15.6.

[85] Jean-Pierre Jouannaud, Claude Kirchner, and Hélène Kirchner. Incremental construction of unification algorithms in equational theories. In Josep Díaz, editor, *Automata, Languages and Programming, 10th Colloquium, ICALP83, Barcelona, Spain, July 18-22, 1983, Proceedings*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer, 1983. Cited in 2, 13.5.1, 14.8, 15.3, 5.

[86] D. Kapur and P. Narendran. Matching, Unification and Complexity. *ACM SIGSAM Bulletin*, 21(4):6–9, 1987. Cited in 14.9.

[87] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988. Cited in 3.1, 7.8, 24.3.

[88] Dohan Kim, Christopher Lynch, and Paliath Narendran. Reviving basic narrowing modulo. In Andreas Herzig and Andrei Popescu, editors, *Frontiers of Combining Systems - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings*, volume 11715 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 2019. Cited in 14.8.

[89] Claude Kirchner and Hélène Kirchner, editors. *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 49, 58.

[90] Nirman Kumar, Koushik Sen, José Meseguer, and Gul Agha. A rewriting based model of probabilistic distributed object systems. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems: 6th IFIP WG 6.1 International Conference, FMOODS 2003, Paris, France, November 19.21, 2003. Proceedings*, volume 2884 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2003. Cited in 1.4.

[91] Joop M. I. M. Leo. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theor. Comput. Sci.*, 82(1):165–176, 1991. Cited in 3.9.

[92] Éduard Lucas. *Recréations mathématiques*. Albert Blanchard, 2 edition, 1992. Cited in 10.

[93] Salvador Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998. Cited in 4.7, 15.5.

[94] Salvador Lucas. Termination of rewriting with strategy annotations. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 669–684. Springer, 2001. Cited in 4.7.

[95] Salvador Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002. Cited in 4.7.

[96] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems — Specification*. Springer, 1992. Cited in 12.

[97] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a strategy language for Maude. In Narciso Martí-Oliet, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2004. Cited in 10.

[98] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a strategy language for Maude. In Narciso Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27–28, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2005. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 21.

[99] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. A rewriting semantics for Maude strategies. In Grigore Roçu, editor, *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008*, volume 238(3) of *Electronic Notes in Theoretical Computer Science*, pages 227–247. Elsevier, 2009. Cited in 10.

[100] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Cited in 7.5.

[101] Ian A. Mason and Carolyn L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 228(1), 1999. Cited in 2.

[102] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. Cited in 1.2, 5.3.

[103] José Meseguer. Multiparadigm logic programming. In Hélène Kirchner and Giorgio Levi, editors, *Algebraic and Logic Programming, Third International Conference, Volterra, Italy, September 2–4, 1992, Proceedings*, volume 632 of *Lecture Notes in Computer Science*, pages 158–200. Springer, 1992. Cited in 15.1.

[104] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993. Cited in 22.1.4, 5.

[105] José Meseguer, editor. *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 32, 57.

[106] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998. Cited in 1.2, 3.5, 4, 4.3, 4.7.

[107] José Meseguer. Strict coherence of conditional rewriting modulo axioms. Technical Report `http://hdl.handle.net/2142/50288`, Computer Science Department, University of Illinois at Urbana-Champaign, August 2014. Cited in 2.

[108] José Meseguer. Order-sorted rewriting and congruence closure. In *Proc. FOSSACS 2016*, volume 9634 of *Lecture Notes in Computer Science*, pages 493–509. Springer, 2016. Cited in 16.6.

[109] José Meseguer. Variant-based satisfiability in initial algebras. *Sci. Comput. Program.*, 154:3–41, 2018. Cited in 16.6.

[110] José Meseguer. Generalized rewrite theories, coherence completion, and symbolic methods. *J. Log. Algebraic Methods Program.*, 110, 2020. Cited in 13.5.2.

[111] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985. Cited in 1.3.

[112] José Meseguer, Joseph A. Goguen, and Gert Smolka. Order-sorted unification. *J. Symbolic Computation*, 8(4):383–413, 1989. Cited in 13.6.

[113] José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Equational abstractions. In Franz Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2003. Cited in 11.4.

[114] José Meseguer and Prasanna Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007. Cited in 13.5.2, 15.1, 15.2, 15.3, 1, 15.4, 15.7.

[115] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979. Cited in 16.6.

[116] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *Journal of the ACM*, 53(6):937–977, 2006. Cited in 16.

[117] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001. Cited in 3.

[118] Vivek Nigam, Carolyn Talcott, and Abraão Aires Urquiza. Towards the automated verification of cyber-physical security protocols: Bounding the number of timed intruders. In *ESORICS 2016: European Symposium on Research in Computer Security*, 2016. Cited in 17.6.12.

[119] Vivek Nigam, Carolyn Talcott, and Abraão Aires Urquiza. Symbolic timed trace equivalence. In *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows*, LNCS, pages 89–111. Springer, 2019. Cited in 17.6.12.

[120] Peter Csaba Ölveczky and José Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002. Cited in 2, 22.2.

[121] Peter Csaba Ölveczky and José Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In T. Margaria and M. Wermelinger, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Held as Part of ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2984 of *Lecture Notes in Computer Science*, pages 354–358. Springer, 2004. Cited in 1.3, 1.4, 1.5.

[122] Peter Csaba Ölveczky and José Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007. Cited in 1.3, 1.4, 1.5, 2, 22.2.

[123] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981. Cited in 21.5.

[124] Gordon Plotkin. Building-in equational theories. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 7, Proceedings of the Seventh Annual Machine Intelligence Workshop, Edinburgh, 1971*, pages 73–90. Edinburgh University Press, 1972. Cited in 13.5.3.

[125] Loic Pottier. Minimal solutions of linear diophantine systems: bounds and algorithms. In Ronald V. Book, editor, *Rewriting Techniques and Applications, 4th International Conference, RTA-91, Como, Italy, April 10-12, 1991, Proceedings*, volume 488 of *Lecture Notes in Computer Science*, pages 162–173. Springer, 1991. Cited in 7.15.

[126] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the 1985 Second Symposium on Logic Programming, Boston, Massachusetts, July 15-18, 1985*, pages 138–151. IEEE Computer Society Press, 1985. Cited in 15.1.

[127] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965. Cited in 13.5.3.

[128] C. Rocha and J. Meseguer. Five isomorphic boolean theories and four equational decision procedures. Technical Report UIUCDCS-R-2007-2818, CS Dept., University of Illinois at Urbana-Champaign, February 2007. Available at `http://hdl.handle.net/2142/11295`. Cited in 1.

[129] Camilo Rocha and José Meseguer. Theorem proving modulo based on boolean equational procedures. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *Relations and Kleene Algebra in Computer Science, 10th International Conference on Relational Methods in Computer Science, and 5th International Conference on Applications of Kleene Algebra, RelMiCS/AKA 2008, Frauenwörth, Germany, April 7-11, 2008.*

Proceedings, volume 4988 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2008. Cited in 13.5.3.

[130] Manfred Schmidt-Schauss. *Computational Aspects of Order-Sorted Logic with Term Declarations*, volume 395 of *Lecture Notes in Computer Science*. Springer, 1989. Cited in 13.6.

[131] Stephen Skeirik and José Meseguer. Metalevel algorithms for variant satisfiability. *J. Log. Algebr. Meth. Program.*, 96:81–110, 2018. Cited in 16.6.

[132] Gert Smolka, Werner Nutt, Joseph Goguen, and José Meseguer. Order-sorted equational computation. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 297–367. Academic Press, 1989. Cited in 13.6.

[133] Mark E. Stickel. A complete unification algorithm for associative-commutative functions. In *Advanced Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, 3-8 September 1975*, pages 71–76, 1975. Cited in 13.7.1.

[134] Carolyn L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998. Cited in 8.1.

[135] Ana Paula Tomás. *On Solving Linear Diophantine Constraints*. PhD thesis, Universidade do Porto, 1997. Cited in 7.15.

[136] Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in Maude 2. In Gadducci and Montanari [76], pages 263–281. `http://www.sciencedirect.com/science/journal/15710661`. Cited in 5.2.

[137] Emanuele Viola. E-unifiability via narrowing. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Theoretical Computer Science, 7th Italian Conference, ICTCS 2001, Torino, Italy, October 4-6, 2001, Proceedings*, volume 2202 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2001. Cited in 13.5.1, 14.8, 21.5.

[138] Patrick Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002. Cited in 1.2, 1.2, 5.3, 5.3, 13.5.3.

Subject Index

Index of Maude Modules

Index of Maude Theories

Index of Maude Views