Maude Manual (Version 2.2)

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

December 2005

Contents

1 Introduction **1**
- 1.1 Simplicity, expressiveness and performance . . . 1
  - 1.1.1 Simplicity . . . 2
  - 1.1.2 Expressiveness . . . 4
  - 1.1.3 Performance . . . 8
- 1.2 The logical foundations of Maude . . . 9
- 1.3 Applications and extensions . . . 12
- 1.4 Core Maude . . . 13
- 1.5 Full Maude . . . 13
- 1.6 Manual structure . . . 13

I Core Maude **17**

2 Using Maude **19**
- 2.1 Getting Maude . . . 19
- 2.2 Running Maude . . . 19
- 2.3 Getting support and more information . . . 23

3 Syntax and Basic Parsing **25**
- 3.1 Identifiers . . . 25
- 3.2 Modules . . . 25
- 3.3 Sorts and subsorts . . . 26
- 3.4 Operator declarations . . . 29
- 3.5 Kinds . . . 30
- 3.6 Operator overloading . . . 31
- 3.7 Variables . . . 32
- 3.8 Terms . . . 33
- 3.9 Parsing . . . 33
  - 3.9.1 Default precedence values . . . 35
  - 3.9.2 Default gathering patterns . . . 35
  - 3.9.3 The extended signature of a module . . . 36
  - 3.9.4 Examples: `parse` . . . 38

4 Functional Modules **41**
- 4.1 (Unconditional) equations . . . 41
- 4.2 (Unconditional) memberships . . . 43
- 4.3 Conditional equations and memberships . . . 43

4.4 Operator attributes . . . 47
4.4.1 Equational attributes . . . 47
4.4.2 The `iter` attribute . . . 48
4.4.3 Constructors . . . 48
4.4.4 Polymorphic operators . . . 49
4.4.5 Format . . . 50
4.4.6 Ditto . . . 53
4.4.7 Operator evaluation strategies . . . 54
4.4.8 Memo . . . 57
4.4.9 Frozen arguments . . . 59
4.5 Statement attributes . . . 60
4.5.1 Labels . . . 60
4.5.2 Metadata . . . 60
4.5.3 Nonexec . . . 60
4.5.4 Otherwise . . . 61
4.6 Admissible functional modules . . . 64
4.7 Matching and equational simplification . . . 65
4.8 More on matching and simplification modulo . . . 67
4.9 Examples: `reduce`, `match`, and `trace` . . . 72

5 System Modules **77**
5.1 (Unconditional) rules . . . 78
5.2 Conditional rules . . . 79
5.3 Admissible system modules . . . 80
5.4 Examples: `rewrite`, `frewrite`, and `search` . . . 82

6 Module Operations **89**
6.1 Module importation . . . 89
6.1.1 Protecting . . . 91
6.1.2 Extending . . . 92
6.1.3 Including . . . 92
6.1.4 Some examples . . . 93
6.2 Module summation and renaming . . . 94
6.2.1 The summation module expression . . . 94
6.2.2 Module renaming . . . 95
6.3 Parameterized programming . . . 97
6.3.1 Theories . . . 98
6.3.2 Views . . . 101
6.3.3 Parameterized modules . . . 105
6.3.4 Module instantiation . . . 108
6.3.5 A specification of sorted lists . . . 113

7 Predefined Data Modules **117**
7.1 Boolean values . . . 117
7.2 Natural numbers . . . 123
7.3 Random numbers and counters . . . 127
7.4 Integer numbers . . . 129
7.5 Rational numbers . . . 132
7.6 Floating-point numbers . . . 136
7.7 Strings . . . 140

7.8 String and number conversions . . . 143
7.9 Quoted identifiers . . . 145
7.10 Basic theories and standard views . . . 146
7.10.1 TRIV . . . 146
7.10.2 DEFAULT . . . 147
7.10.3 TAO-SET . . . 148
7.11 Containers: lists and sets . . . 149
7.11.1 Lists . . . 149
7.11.2 Sets . . . 152
7.11.3 Relating lists and sets . . . 154
7.11.4 Generalized lists . . . 155
7.11.5 Generalized sets . . . 157
7.11.6 Sortable lists . . . 160
7.12 Maps and arrays . . . 162
7.12.1 Maps . . . 162
7.12.2 Arrays . . . 164
7.13 A linear Diophantine equation solver . . . 165

8 Object-Based Programming 169
8.1 Configurations . . . 169
8.2 Object-message fair rewriting . . . 179
8.3 Example: data agents . . . 181
8.4 External objects . . . 186
8.4.1 Buffered sockets . . . 193

9 Model Checking 197
9.1 LTL formulae and the LTL module . . . 197
9.2 Associating Kripke structures to rewrite theories . . . 199
9.3 LTL model checking . . . 203
9.4 The LTL satisfiability and tautology checker . . . 209

10 Reflection, Metalevel Computation, and Strategies 211
10.1 Reflection and metalevel computation . . . 211
10.2 The META-TERM module . . . 212
10.2.1 Metarepresenting sorts and kinds . . . 212
10.2.2 Metarepresenting terms . . . 213
10.3 The META-MODULE module: Metarepresenting modules . . . 214
10.4 The META-LEVEL module: Metalevel operations . . . 218
10.4.1 Moving between reflection levels: upModule, upTerm, downTerm, and others 218
10.4.2 Simplifying and rewriting: metaReduce, metaRewrite, and metaFrewrite 221
10.4.3 Applying rules: metaApply and metaXapply . . . 223
10.4.4 Matching: metaMatch and metaXmatch . . . 226
10.4.5 Searching: metaSearch and metaSearchPath . . . 228
10.4.6 Parsing and pretty-printing: metaParse and metaPrettyPrint . . . 230
10.4.7 Sort operations . . . 232
10.4.8 Other metalevel operations: wellFormed . . . 239
10.5 Internal strategies . . . 240

11 User Interfaces and Metalanguage Applications **245**
11.1 The LOOP-MODE module . . . 245
11.2 User interfaces . . . 246
11.3 Using the loop . . . 249
11.4 Tokens, bubbles, and metaparsing . . . 250

12 Debugging and Troubleshooting **257**
12.1 Debugging approaches . . . 257
12.1.1 Tracing . . . 257
12.1.2 Term coloring . . . 258
12.1.3 The debugger . . . 259
12.1.4 The profiler . . . 259
12.1.5 Performance note . . . 260
12.2 Traps and known problems . . . 260
12.2.1 Associativity and idempotence . . . 260
12.2.2 Segmentation fault (core dumped) . . . 260
12.2.3 Bare variable lefthand sides . . . 260
12.2.4 Operator overloading and associativity . . . 261
12.2.5 Preregularity and associativity . . . 261
12.2.6 Collapse theories . . . 263
12.2.7 One-sided identities and associativity . . . 264
12.2.8 Memberships for associative operators . . . 265
12.2.9 Memberships for iterated operators . . . 268

II Full Maude **271**

13 Full Maude Extensions **273**
13.1 Running Full Maude . . . 274
13.2 Using Core Maude modules in Full Maude . . . 278
13.3 Additional module operations in Full Maude . . . 279
13.3.1 The n-tuple module expression . . . 281
13.3.2 Parameterized views . . . 282
13.4 Moving up and down between reflection levels . . . 285
13.4.1 Up . . . 285
13.4.2 Down . . . 287
13.5 Differences between Full Maude and Core Maude . . . 288

14 Object-Oriented Modules **291**
14.1 Object-oriented systems . . . 292
14.1.1 Objects and messages . . . 292
14.1.2 Classes . . . 292
14.1.3 Inheritance . . . 293
14.1.4 Object-oriented rules . . . 294
14.2 Example: a rent-a-car store . . . 296
14.3 Object-oriented parameterized programming . . . 300
14.3.1 Theories . . . 300
14.3.2 Views . . . 300
14.3.3 Parameterized modules . . . 301
14.4 Module operations on object-oriented modules . . . 303

14.4.1 Module summation and renaming . . . 303
14.4.2 Module instantiation . . . 304
14.5 Example: extended rent-a-car store . . . 305
14.6 A strategy for sequential rule execution . . . 309
14.7 Model-checking a round-robin scheduling algorithm . . . 313
14.8 From object-oriented modules to system modules . . . 317

III Reference 321

15 Complete List of Maude Commands 323
15.1 Command line flags . . . 323
15.2 Rewriting commands . . . 324
15.3 Matching commands . . . 326
15.4 Searching commands . . . 326
15.5 Tracing commands . . . 327
15.6 Print option commands . . . 328
15.7 Show option commands . . . 328
15.8 Show commands . . . 329
15.9 Profiler commands . . . 330
15.10 Debugger commands . . . 330
15.11 Miscellaneous commands . . . 330
15.12 System commands . . . 331

16 Core Maude Grammar 333
16.1 The grammar . . . 333
16.2 Synonyms . . . 337
16.3 Lexical Issues . . . 338

Bibliography 344

Chapter 1

Introduction

This introduction tries to give the big picture on the goals, design philosophy, logical foundations, applications, and overall structure of Maude 2. 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 and as easy to understand as possible, and should have a clear semantics.

- *Expressiveness*: it should be possible to naturally express a very wide range of applications, ranging from deterministic systems to highly concurrent ones, and encompassing both code and specifications. *Semantic framework* uses, in which not just applications, but entire formalisms (other languages, other logics) can be naturally expressed should also be supported.

- *Performance*: it should be possible to use the language not only for executable specification and system prototyping, but as a real programming language with competitive performance.

Although simplicity and performance are in principle natural allies, maximizing at the same time the expressiveness dimension is perhaps the key point in Maude's language design. Languages are after all *representational devices*, and it is on how generally, naturally, and easily problems and applications can be represented and can be reasoned about that the merits of a language should be judged. Of course, *domain-specific* languages have also an important role to play in different application areas, and can offer a useful "economy of representation" for a given area. In this regard, Maude should be viewed as a *metalanguage*, through which many different domain-specific languages can be easily 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 (quoted identifiers are elements of 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 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 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 for equals' use of equations in elementary algebra and has a very clear and simple semantics in equational logic. Replacement of equals for 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, if it exists, 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

[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).

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

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 both equations and rules 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 (`Accnt`) and a `bal`(ance) attribute, say, a natural number. The following then are two different account objects in our notation:

```
< 'A-001 : Accnt | bal : 200 >
< 'A-002 : Accnt | 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 : Accnt | bal : 200 >
< 'A-002 : Accnt | 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 : Accnt | bal : M > credit(I, N)
  => < I : Accnt | bal : (M + N) > .

crl < I : Accnt | bal : M > debit(I, N)
  => < I : Accnt | 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 concurrently be applied to different fragments of the soup. For example, applying both rules to the soup above we get the new distributed state:

```
< 'A-001 : Accnt | bal : 175 >
< 'A-002 : Accnt | 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 Chapter 4 and Section 1.1.2 below). 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 executed, a bank account can end up in quite different states. 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. Indeed, system modules extend the purely functional semantics of equations to the concurrent rewriting semantics of rules.[3] Although certainly declarative, 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,

[2] This sublanguage is essentially an extension of the OBJ3 equational language [38], 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.

- generic types and modules,
- support for objects, and
- 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 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 : Accnt | 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 : Accnt | 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 makes readability (and therefore understandability) of programs and data much easier. 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 [3]. 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.

[4] Except for the combination of associativity and idempotency, which is not supported currently.

Types, subtypes, and partiality

Maude has two kinds 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

```
Nat < Int < Rat < Cpx < Quat
```

indicating that the naturals are contained in the integers, and these are contained in the rationals, which are contained in the complex numbers, which are in turn contained in the quaternions. 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 result has a kind but not a sort should be considered *undefined* for that argument. 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 defining 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):

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

Generic types and modules

Maude supports a powerful form of *generic programming* that substantially extends the *parameterized programming* capabilities of OBJ3 [38]. 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 `POSET` of partially ordered sets (or perhaps `TOSET` if the order is assumed total) 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. The analogous of dependent types is also supported by making the parameter instantiations depend on specific constants in the parameter theory, and by giving membership axioms depending on such constants. For example, natural numbers modulo n, 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 13.3.1) 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. Finally, *parameter theories can themselves be parameterized*, yielding a general analogue of what de Bruijn calls "telescopes." For example, a tensor algebra module can be parameterized by the theory of vector spaces, which is itself parameterized by the theory of fields.

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 syntax in *object-oriented modules*. Such modules directly support object-oriented concepts like objects, messages, classes, and multiple class inheritance. Although not yet supported by the current version, future releases will also support *communication with external objects*. This will allow Maude objects to interact by message passing with all kinds of external objects, such as files, databases, graphical user interfaces, internet sockets, sensors, robots, and so on; and will permit programming with rewrite rules such interactions. It will also easily allow *distributed implementations* of Maude, in which a "soup" of objects and messages will not be realized anymore 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 10 and [19, 20]). 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 for a given application we want to use a different strategy? The answer is that Maude modules can be executed at the metalevel with user-definable *internal strategies*.[5] 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).

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

- *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, Church-Rosser and termination checkers, Knuth-Bendix completion tool, the Real-Time Maude tool, and so on.

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 formulae 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 (the above rough figures assume a 900 Mhz Pentium). Similarly, associative and associative-commutative equational rewriting with term patterns used in practice[6] can be performed at the typical rate of one 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 would have millions of elements. This situation has been drastically altered by a recent result of Steven Eker [33] 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 2 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 [31] 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

[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.

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 rules are most expensive to execute in a given application (see Section 12.1.4). Another is *evaluation strategies* (see Section 4.4.7), giving the user the possibility of indicating which arguments to evaluate before simplifying a given operator with the equations. This can range from "no 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 10.4). This, together with 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. It 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* [47, 6] is embedded in *rewriting logic* [45, 8].

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, $(\Sigma, E \cup A, \phi_0, \emptyset)$, where ϕ_0 makes all arguments frozen for all operators. 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 whatever freezing information he/she wants to each operator in the signature of a functional module, not just the ϕ_0 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 (or should have) 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 $(\Sigma, E \cup A)$ 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 data in the states of this system is provided by the underlying initial algebra $T_{\Sigma/E \cup A}$. The labeled state transitions are the (possibly complex) *concurrent rewrites* possible in the system by application of the rules R. For our bank accounts example, these labeled 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, or should have, in mind.

How do the mathematical models associated to 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 impressionistic (see Chapter 4 and [6] for the whole story in the case of functional modules, and Chapter 5 and [64] 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

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

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)))
```

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 [64]. 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.

All these observations on the mathematical and operational semantics of Maude programs are of enormous importance for reasoning about them, and verifying their correctness. First of all, note that 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, by giving some statements in a Maude module the `nonexec` attribute (see Section 4.5.3), we can even integrate *nonexecutable* specifications. This can be very useful in several ways. For example, we may include *lemmas* that we have proved about a module as nonexecutable statements. Similarly, we may begin with nonexecutable specifications for a module, and then transform them into equivalent ones that are executable.

The essential point is that, unlike conventional code, a Maude module is by construction a *mathematical object*. Therefore, we can directly use all the tools of mathematics and logic, including automatic or semiautomatic tools, to reason about the correctness of Maude modules. The Maude interpreter itself is the first and most obvious such tool. In fact it is a high-performance *logical engine* to prove logical facts about our theories. Furthermore, as already mentioned, Maude has a collection of formal tools supporting different forms of logical reasoning to verify program properties, including:

- a model checker to verify linear temporal logic (LTL) properties of finite-state system modules;
- an inductive theorem prover (ITP) to verify properties of functional modules;
- a Church-Rosser checker, to check such a property for functional modules;
- a Knuth-Bendix completion tool and termination checker for functional modules; and

- a coherence checker for system modules.

Several of these tools have been or will soon be ported from earlier versions of Maude to Maude 2. The LTL model checker is part of the Maude 2 distribution and is described in detail in Chapter 9.

1.3 Applications and extensions

Given the generality and expressiveness of Maude, it is futile to try to come up with an exhaustive list of applications. One way to gain some feeling for the wide range of applications that are possible is to consult the over 300 papers on Maude's underlying logic (rewriting logic) and its many applications, many of them developed in Maude [43]. An obvious general remark is that applications that naturally fit within the functional, resp. the object-oriented, paradigms will typically be very easy to express in Maude as functional, resp. object-oriented, modules.

Abandoning any attempt to be exhaustive, it is nevertheless possible to mention a few sample areas where Maude seems to be particularly well-suited to program applications:

- *Bioinformatics*. Cell reactions can easily be expressed as rewrite rules. This allows easy development of models of cell biology that can be used, in conjunction with Maude's LTL model checker, to analyze and predict cell behavior [34, 35].
- *Hardware, software, and networks* executable specification and analysis (see [43, 48]).
- *Meta-tool applications*, in which Maude is used to develop a wide variety of software transformation, analysis, and formal methods tools (see [17, 43]).
- *Programming language definitions*. The operational semantics of a programming language can be easily described by rewrite rules [42, 7, 62]. Such executable semantic definitions can then be used to perform different formal program analyses [56, 10].
- *Logical framework applications*. Many different logics can be both represented and mechanized in Maude as rewrite theories [42, 43]. Furthermore, it is possible to perform metalogical reasoning about such logics using reflection and Maude's inductive theorem prover [2].

As soon as support for communication with external objects becomes available in Maude, many other application areas such as:

- internet programming,
- mobile computing,
- distributed agents, and
- reflective middleware

will be ripe for system development in Maude; for the time being the Maude interpreter has been applied in these areas primarily for prototyping and formal specification purposes.

There are also a number of *extensions* of Maude, written in Maude, that target specific application areas. Full Maude is the module algebra extension documented in Part II of this manual. The Real-Time Maude tool [50, 51] supports specification and formal analysis of real-time and hybrid systems. The PMaude prototype [40] supports specification and execution of probabilistic systems. The BMaude language design [49] extends Maude to the area of behavioral specifications.

1.4 Core Maude

We call *Core Maude* the Maude 2 interpreter implemented in C++ and providing all of Maude's basic functionality. Part I of this manual 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, and debugging and troubleshooting.

1.5 Full Maude

Full Maude is an extension of Maude, written in Maude itself, that endows the language with a more powerful and extensible *module algebra* in which Maude modules can be combined together to build more complex modules. 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 objects, messages, classes, and inheritance. It is also possible to move up and down the reflective tower using Full Maude commands.

1.6 Manual structure

The present manual documents Maude 2, 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 `http://maude.cs.uiuc.edu`.

The document 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 sketch of what can be found in the remaining chapters:

Part I. Core Maude.

Chapter 2 is a brief introduction on 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 constructs of the language, including what is an identifier, a sort, or 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, 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 is a fundamental issue.

Chapter 5 introduces system modules, and is mainly devoted to rules and term rewriting.

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 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 c 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. It also describes how communication with external objects is supported in Core Maude.

Chapter 9 introduces the facilities for model-checking provided by the Maude system. After introducing linear temporal logic, it explains how model checking can be used to prove properties of a system module.

Chapter 10 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 a discussion on internal strategies.

Chapter 11 explains the way of using the facilities provided by the modules `META-LEVEL` and `LOOP-MODE` for the construction of user interfaces and metalevel applications.

Chapter 12 discusses debugging and troubleshooting, considering the different facilities for debugging provided: tracing, term coloring, the debugger, and the profiler. A number of traps and known problems are also given.

Part II. Full Maude.

Chapter 13 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 (with tuple generation and parameterized views), and on the facilities available in Full Maude for moving up and down between reflection levels.

Chapter 14 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 15 gives a complete list of the commands available in Maude.

Chapter 16 includes the grammar of Core Maude.

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 Christiano Braga, Grit Denker, Joe Hendrix, Merrill Knapp, Nirman Kumar, Peter Ölveczky, Adrián Riesco, Dilia Rodriguez,

Grigore Roşu, 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 Peter Ölveczky, Miguel Palomino, Sylvan Pinsky, Isabel Pita, Manuel Roldán, Antonio Vallecillo, and Alberto Verdejo for their comments on previous versions of this document.

We are grateful to José Quesada, who developed MSCP, the Maude parser. MSCP is implemented using SCP [54] as the formal kernel, and provides a basis for flexible syntax definition, and an efficient treatment of what might be called *syntactic reflection*.

As already mentioned, Maude's historical precursor is the OBJ3 language [38]. 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 [5] and CafeOBJ [37], 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, and Toshimi Sawada.

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 (shown in Figure 2.1) `http://maude.cs.uiuc.edu`. There you can also find documentation about Maude, including a Maude tutorial, some papers on Maude and rewriting logic, and several Maude applications, including a set of proving tools for Maude specifications and Maude case studies.[1]

Maude binaries are provided for selected architectures and operating systems. You can find information on this in the Maude web site. There you can also find installation instructions.

2.2 Running Maude

You can start a session with Maude by calling the `maude.linux` binary in the `maude-linux/bin` directory in a Linux shell window (and similarly for other platforms):

```
maude-linux/bin$ maude-linux
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.2 built: Oct  7 2005 18:22:06
             Copyright 1997-2005 SRI International
                   Tue Oct 18 23:41:05 2005
Maude>
```

The Maude system is now ready to accept Maude modules and commands (see Chapter 15 for a complete list of Maude commands). During a Maude session, the user interacts with the system by introducing his/her request at the Maude prompt. For example, one can quit:

```
Maude> quit
```

`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 `MY-NAT`, which specifies the natural numbers in the Peano notation with a plus operation `_+_` on them.

[1] Most of the material in the web is still on Maude 1. We plan to update it in the near future.

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

```
Maude> fmod MY-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` or `load` commands. Assuming that the file `my-nat.maude` contains the module `MY-NAT` above, we can do the following to enter it:

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

After entering `MY-NAT` we can, for example, reduce the term `s s zero + s s s zero` (which corresponds to $2 + 3$ in the Peano notation) in it.

```
Maude> reduce in MY-NAT : s s zero + s s s zero .
reduce in MY-NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
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 15.11).

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

Although it is not the case in this simple example, sometimes we get a very big term as output from Maude and then we tabulate its presentation in this manual, in order to make it easier to read and understand.

When you execute `maude.linux`, the file `prelude.maude`, which includes several predefined modules (see Chapter 7), is automatically loaded. To find `prelude.maude` Maude checks, in order,

(a) the directories specified in the `MAUDE_LIB` environment variable,

(b) the directory containing the executable, and

(c) 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`-uce request:

```
Maude> red in STRING : "hello" + " " + "world" .
reduce in STRING : "hello" + " " + "world" .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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) (~ rewrites/second)
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,

(a) the current directory,

(b) the directories in the `MAUDE_LIB` environment variable, and

(c) 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, it is common among Maude users to use 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 we wish. When entering it with an `in` or `load` command, Maude will read them one after another as if they were written 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 (whether backquoted or not) must balance within multiline comments.

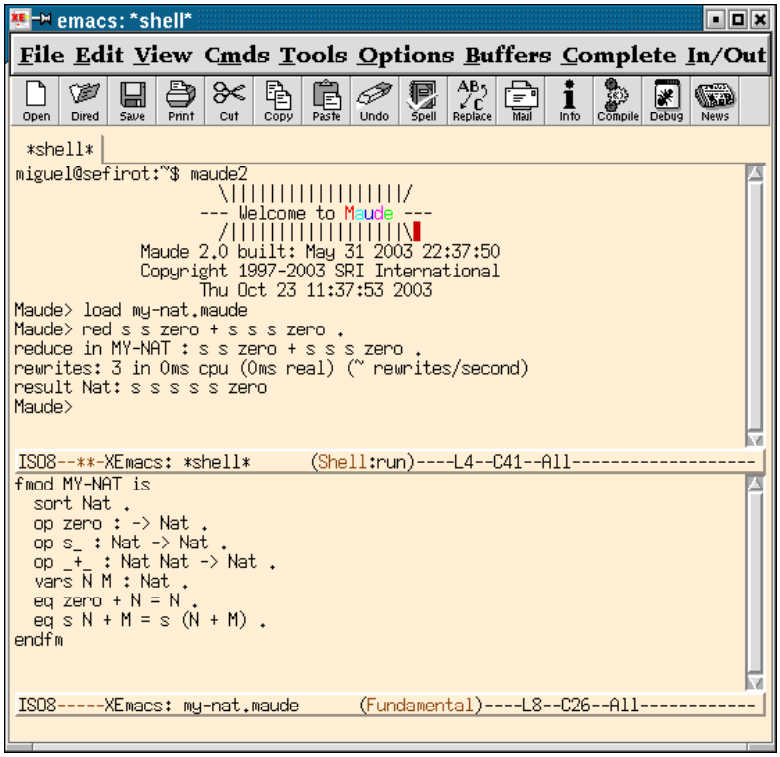


Figure 2.2: Running Maude inside Emacs.

2.3 Getting support and more information

We maintain the following mailing lists related to Maude:

- `maude-users@maude.cs.uiuc.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

  `http://maude.cs.uiuc.edu/mailman/listinfo/maude-users/`

- `maude-help@maude.cs.uiuc.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

  `http://maude.cs.uiuc.edu/pipermail/maude-help/`

- `maude-bugs@maude.cs.uiuc.edu`. A list for reporting any problems you experience with Maude, and also any suggestions for enhancements and improvements. Please, when

submitting a bug make sure you provide information of the concrete release of Maude (and Full Maude if it is the case) and the platform on which you experimented the problem, in addition of course to the information on the bug itself.

Chapter 3

Syntax and Basic Parsing

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 string 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 nonempty 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 [39, Section A2.5.2].

3.2 Modules

In Maude the basic units of specification and programming are called *modules*. In Core Maude there are two kinds of modules: *functional modules* and *system modules*.

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* 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 composed 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* 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 composed names. For example, a module specifying 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 what follows we will describe the syntactic elements common to both functional and system modules, such as sorts, subsorts, operators, kinds, variables, and terms, postponing the discussion of 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 are partially ordered via a *subsort* relation.

A sort is declared using the keyword `sort` 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 keyword `sorts`, 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.

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

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. Also, 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 10.2.1. The use of '`{`', '`}`', and '`,`' is only allowed in structured sorts (see below). Although any so restricted identifier is a legal sort name, the preferred style is to capitalize the *first* letter of the name. Also, in the case of a composed 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 can also be structured. 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⟩     ::= ⟨regular sort name⟩
             | ⟨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}           (no regular sort name prefix)
a(X, Y)       (',' not inside braces)
a{b, {d}}{e}  ({d} lacks regular sort name 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 10).

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 regular sort names.

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⟩ .
```

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

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

say 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:

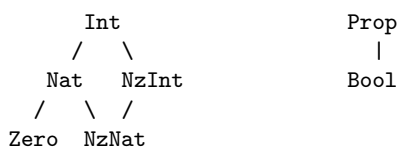
```
subsorts Zero NzNat < Nat .
```

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 defines 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, and then the connected components are exactly those of the underlying graph.



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*), followed by `->`, followed by the sort of its result (called the operator's *coarity*), 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. 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] .
```

`assoc` is an attribute declaring that sequence concatenation is associative (see Section 4.4). 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 nonempty 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.[1]

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 another 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`.

3.5 Kinds

The equational logic underlying Maude is membership equational logic [47, 6]. In this logic sorts are grouped into equivalence classes called *kinds*. For this purpose, two sorts are considered equivalent 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 set 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 ok; otherwise, the fully simplified error expression is returned

[1] 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.

as an error message. Equational simplification can also occur at the kind level, 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 are supposed to have the same behavior on their shared argument sort, that is, the operator on the subsort is the

restriction of the operator on the larger sort. 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).

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.

In particular, this rules out ad-hoc overloaded constants. Therefore, 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

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 $f : s_1 \ldots s_n \rightarrow s$, 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 desirable property of a module, called *preregularity*, is that each term has a *least* sort that can be assigned to it. Maude assumes that modules are preregular, and generates warnings when a module is loaded if the property does not hold.

A *ground term* is a term not containing any 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 different functions should be applied to data (for example, `(s zero) + (s zero)`). Ground terms constitute the term algebra associated to 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 using a bison/flex based parser for Maude's surface syntax, a grammar generator which generates the context-free grammar for the user-defined mixfix parts of a Maude module over the user's signature, and the MSCP context-free parser (generator) that generates a parser for the context-free grammar. MSCP was developed by J. Quesada [55, 54].

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:

```
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,[2] 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: 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.

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)`, which makes them to be applied only on terms of smaller (or equal) precedence value. Since the precedence of a term is given by the precedence of its top operator, `(1 + 2) * 3` is not a valid parse because the term `1 + 2` has precedence value 33, which is greater than the precedence of `_*_`. Note that by default all constants have precedence 0 (see Section 3.9.1), and therefore they are also valid arguments for both operators.

The gathering pattern of an operator restricts the precedences of terms that are allowed as arguments. We give a (nonempty) 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

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

- `&` indicates that the operator allows any precedence value for the corresponding argument.

With gathering pattern `(E E)`, the arguments of `_+_` can be terms with its same precedence value, and thus `1 + 2` and `2 + 3` are valid arguments. 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 parentheses can appear as argument of any term.

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 [38]. 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 associates 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 [38]. 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. For example, in the module `META-MODULE` (see Section 10.3), 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`. Additional examples can be found e.g. in Section 3.6.

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

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

  for each sort `S` in its signature.[3] 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)`. Also, 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))`.

- *Polymorphic operators and the* `BOOL` *module.* All the information contained in the predefined modules `TRUTH-VALUE`, `TRUTH`, and `BOOL` is included in the extended signature of each module. 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`, sort predicates `_:: S` and `_::: S` are also added. If the `IDENTICAL` module is imported, then the polymorphic operators `_===_` and `_=/==_` are also added (see Section 7.1).

[3] Note that, as said in Section 3.4, an extra pair of parentheses is required for such a declaration to be correct.

3.9.4 Examples: `parse`

Maude provides the `parse` command for parsing terms. The command does not do anything more than parsing in the extended signature of the module, that is, exactly what is done first for any other command. For example, when we try to reduce a term `(2 + 3) * 5`, the system 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 96: didn't expect token true: 2 + true <---*HERE*
Warning: <standard input>, line 96: no parse for term.
```

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

```
Maude> parse in NAT : 2 + true .
Warning: <standard input>, line 97: didn't expect token true: 2 + true <---*HERE*
Warning: <standard input>, line 97: no parse for term.
```

As other commands, parsing can take place on the default module, or in the module specified in the command. Its syntax is as follows:

`parse` [`in` ⟨*ModId*⟩ `:`] ⟨*Term*⟩ `.`

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

```
fmod MY-NAT-1 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 MY-NAT-1 : 1 + (2 * 3) .
Nat: 1 + (2 * 3)

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

Let us now consider the module `MY-NAT-2`, where `_+_` and `_*_` are declared with precedences 33 and 31, respectively.

```
fmod MY-NAT-2 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 MY-NAT-2 : 1 + 2 * 3 .
Nat: 1 + 2 * 3
```

Of course, we may still use parentheses.

```
Maude> parse in MY-NAT-2 : (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 MY-NAT-2 : 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 MY-NAT-2 : (1 + 2) + 3 .
Nat: (1 + 2) + 3

Maude> parse in MY-NAT-2 : 1 + (2 + 3) .
Nat: 1 + (2 + 3)
```

Let us now consider the module `MY-NAT-3`, where `_+_` and `_*_` are declared to be left-associative, that is, with gathering patterns `(E e)`.

```
fmod MY-NAT-3 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 parsings.

```
Maude> parse in MY-NAT-3 : 1 + 2 * 3 .
Nat: 1 + 2 * 3

Maude> parse in MY-NAT-3 : 1 + 2 + 3 .
Nat: 1 + 2 + 3
```

Let us now consider the module `MY-NAT-4`, where `_+_` and `_*_` are declared to be associative. Note that in this case, by default, they are assigned gathering patterns `(E e)`.

```
fmod MY-NAT-4 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 MY-NAT-4 : 1 + 2 * 3 .
Nat: 1 + 2 * 3

Maude> parse in MY-NAT-4 : 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 MY-NAT-1 : (2 + 3).Nat .
Nat: 2 + 3

Maude> parse in MY-NAT-1 : (2).Nat + 3 .
Nat: 2 + 3

Maude> parse in MY-NAT-1 : (2).Nat + (3).Nat .
Nat: 2 + 3

Maude> parse in MY-NAT-1 : ((1) + ((2) + (3))) .
Nat: 1 + (2 + 3)

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

Maude> parse in MY-NAT-4 : _+_(1, 2, 3) .
Nat: 1 + 2 + 3

Maude> parse in MY-NAT-4 : if 1 == 2 then 1 + 2 else _+_(1, 2) fi .
Nat: if 1 == 2 then 1 + 2 else 1 + 2 fi

Maude> parse in MY-NAT-4 :
         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 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.

The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic called *membership equational logic* [47, 6]. 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 functional modules (see Chapter 6), 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

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 a 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:

```
vars N M : Nat .
eq N + zero = N .
eq N + s M = s (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 execute them (except at the metalevel with a user-given strategy, see Section 10.5); 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

Membership axioms 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 [ctor] .
  op s_ : Nat -> Nat [ctor] .

  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 12.2.8 and 12.2.9.

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 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 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,

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

[2] Note that the `source` and `target` operations can also be declared, for example, as

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

[3] Similar constructs are used in languages like ASF+SDF [61], and ELAN [5].

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 instead 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

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

[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 29, where it is declared to be associative. If we consider the declaration of this operator given in page 47, 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`.

does not reduce further.

Matching equations in conditions give great expressive power and consequently some care is needed in using matching equations in conditions 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.

```
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` depending on the matching. 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 unique 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 numbers for a two.

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* [47, 6]. 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).

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's 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 exists [47], 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 15.2 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 attribute 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).

These attributes are only allowed for *binary* operators such that the argument sorts and the result sort all belong to the same connected component. An operator can be declared with any combination of these attributes, which may appear in any order in the attribute declaration. The only restriction is that the `idem` attribute *cannot be used together with the* `assoc` attribute, since the combination of these two attributes is not currently supported by the Maude implementation.

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, for example `x + y = y + x`, is declared as an ordinary equation, then it will easily produce looping, nonterminating simplifications. If it is 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 for 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.

Notice that matching modulo associativity and membership axioms can interact in undesirable ways, as explained in Section 12.2.8.

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 be 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 and iterated operators may also interact in undesirable ways; see Section 12.2.9 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, a 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 12) 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 __ : NatSeq NatSeq -> NatSeq [ctor assoc] .
```

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 typically will *not* be a constructor. Indeed, we can define the sort `Int` with a subsort `NzNeg` of nonzero negative numbers built up with a "minus" constructor, and we can then specify both minus and successor 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
```

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  .
```

4.4.4 Polymorphic operators

A number of Maude's built-in operators are *polymorphic* in one or more arguments in the sense that they have meaning when these arguments are of any known sort. Examples include Boolean

[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.

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. Polymorphs 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 7.8 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 among characters 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 10).

- 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 DOS windows, but *not* Emacs shell buffers, unless you use `ansi-color.el` (there is a copy of this Emacs Lisp file with the Maude distribution just in case your Emacs distribution lacks it).

`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:[7]

```
Maude> reduce red green blue yellow cyan magenta .
reduce in COLOR-TEST : red green blue yellow cyan magenta .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
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 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 -> Bool .

  op while_do_od : Bool 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

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

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 9, or in the modules `META-TERM` and `META-MODULE` (in the file `prelude.maude`), described in Chapter 10.

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). 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 the previous subsort-overloaded version, except for the `ctor` attribute, which is 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`. That is, an operator given the `ditto` attribute either has no other explicitly given attributes, or has also only the `ctor` attribute if it is a constructor. 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 9.

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 generaly, 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 controling 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 [38] 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 [32].

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 $(12...n)$; this is the "eager evaluation" case. The opposite extreme is a form of lazy evaluation such as the lazy append operator in `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 (i1 ... ik 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
  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] .
reduce in EXT-BOOL : false and-then b:[Bool] .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
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, that we have displayed side-by-side to emphasize their only difference, namely, the evaluation strategy associated to the operator `g`.

```
fmod STRAT1 is
  sort S .
```

```
fmod STRAT2 is
  sort S .
```

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

The canonical form of the term `g(a)` in `STRAT1` is `g(b)`, but in `STRAT2` 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. `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 15.2). For example, to compute the list of the first ten prime numbers we evaluate the expression:

```
Maude> reduce show primes upto 10 .
reduce in SIEVE : show primes upto 10 .
rewrites: 415 in 0ms cpu (0ms real) (~ rewrites/second)
result NatList: 2 . 3 . 5 . 7 . 11 . 13 . 17 . 19 . 23 . 29
```

The paper [32] documents the operational semantics and the implementation techniques for Maude's operator evaluation strategies in much more detail.

Of course, operator evaluation strategies, while quite useful, are by design restricted in their scope of applicability to *functional modules*.[8] 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 10.5). 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

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

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 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 FIBO 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 FIBO : fibo(50) .
rewrites: 61095033220 in 132081000ms cpu (145961720ms real) (462557 rewrites/second)
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 FIBO : fibo(50) .
rewrites: 148 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 12586269025

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

Maude> red fibo(1000) .
reduce in FIBO : fibo(1000) .
rewrites: 2701 in 0ms cpu (11ms real) (~ rewrites/second)
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 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 .
```

`set clear memo` is off by default.

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 until 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.

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 four attributes are available: `label`, `metadata`, `nonexec`, and `owise`. The attributes `label`, `metadata`, and `nonexec` 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 been written

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

4.5.2 Metadata

The `metadata` attribute must be followed by a string (that is, a data element in the `STRING` module, see Section 7.7). 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 the declaration and not the operator. 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 FOO 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 non-executable as follows.

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

Although non-executable 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[9].

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

```
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 be against any constructs that violate the logical semantics of the language.

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 $enabled_f$ 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[10] is the interaction between `owise` equations and the operator evaluation strategies discussed

[9] Indeed, several languages have conventions of this kind, including ASF+SDF [61].

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

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 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

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

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 $enabled_f : [s_1] \dots [s_n] \to \texttt{[Bool]}$, with the same evaluation strategy as f. Note the somewhat subtle use of the matching equations $enabled_f(v_1^j, \dots, v_n^j) := enabled_f(y_1, \dots, y_n)$, $1 \leq j \leq k$, in the conditions. Since f and $enabled_f$ have the same strategy, after the arguments of the matching instance of the expression $enabled_f(y_1, \dots, y_n)$ become evaluated according to the strategy, we are then able to

match $enabled_f(v_1^j, \ldots, v_n^j)$ 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 $f(v_1^j, \ldots, v_n^j)$ to the same subject term after its subterms under f had been evaluated according to f's strategy.

Of course, the semantics of the $enabled_f$ predicate is defined in the expected way by the equations

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

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

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

The result is `g(c)` (in spite of `g` having strategy 0) because the `owise` equation is not considered until after evaluating the final 0 in the strategy, and by then `b` is simplified to `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.

4.6 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 *least sort possible* in the subsort ordering.

The traditional requirement in this context is that, given a conditional equation[11] $t = t'$ *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:

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 [6, 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 have 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, and E the (possibly conditional) equations and memberships specified as statements.

Ground terms in the signature Σ form a Σ-algebra denoted T_Σ. 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. Similarly, equivalence classes of terms

[11] 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 $true = true$.

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.

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 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 less 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 function on terms $\sigma : T_\Sigma(X) \longrightarrow T_\Sigma$ that we denote with the same name.

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*[12] 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 p of t such that l matches $t|_p$ via a well-sorted substitution[13] σ and t' is obtained from t by replacing the subterm $t|_p \equiv \sigma(l)$ with the term $\sigma(r)$. Furthermore, 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* (also called *equational rewriting*[14]) by $t \rightarrow_E t'$, where the possible equations for rewriting are chosen in the set E. The reflexive and transitive closure of the relation $\rightarrow_E$ is denoted $\rightarrow_E^*$.

A set of equations E is *confluent* when any two rewritings of a term can always be unified by further rewriting: if $t \rightarrow_E^* t_1$ and $t \rightarrow_E^* t_2$, then there exists a term t' such that $t_1 \rightarrow_E^* t'$ and $t_2 \rightarrow_E^* t'$.

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

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 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: $t\downarrow_E \equiv t'\downarrow_E$.

[12] Our terminology is the opposite of the one used by some authors, who would instead say that u matches t.

[13] 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.

[14] 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.

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 [6].

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 must be generalized to equational simplification, confluence, and termination *modulo* A. We discuss this in more detail in Section 4.8 below.

Equational specifications in Maude functional modules (and in the equational part of system modules) are assumed to be (ground[15]) Church-Rosser and terminating, and their operational semantics 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 [26, 23].

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
```

[15] That is, the Church-Rosser and terminating properties hold for all ground terms.

```
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 [22].

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,

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

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

A second source of unexpected behavior is the fact that a lefthand side involving an associative 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

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

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 `X` $\mapsto$ `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 for just this purpose:

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

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

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

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 Examples: `reduce`, `match`, and `trace`

Here we assemble the `NUMBERS` running example to illustrate some of the basic commands for interacting with Maude. For full details about these and other Maude commands see Chapter 15.

```
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 .
  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 sd(N, N) = zero .
  eq sd(N, zero) = N .
  eq sd(zero, N) = N .
  eq sd(s N, s M) = sd(N, M) .
  eq p(s N) = N [label partial-predecessor].
  eq N + zero = N .
  eq N + s M = s (N + M) .

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

  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 .
  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. 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 .
reduce in NUMBERS : s zero + s zero + s zero .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: s s s zero

Maude> red 1 + (1 + 1) .
reduce in NUMBERS : 1 + (1 + 1) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat3: 0
```

The next example illustrates the effect of the idempotency equation for natural number sets

```
Maude> red zero ; s zero ; zero ; s zero .
reduce in NUMBERS : zero ; s zero ; zero ; s zero .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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) (~ rewrites/second)
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) (~ rewrites/second)
result Bool: false
```

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) (~ rewrites/second)
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 and hence there is only one occurrence of the subterm `p(s zero)` in the internal representation.

We can ask Maude to show the module `FIBO` (assuming it has been loaded).

```
Maude> show module FIBO .
fmod FIBO 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 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`.

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.

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`, and `nonexec` (see Section 4.5). The `owise` attribute is only used with equations.

5.1 (Unconditional) rules

Mathematically, a rewrite rule has the form $l : t \to t'$, with t, t' terms of the same kind, which may contain variables. 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.

A 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. 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. 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.

```
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 is used to print the constants using different colors, so that coins can easily be separated from items in a given marking (see Section 4.4.5).

```
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, which can be represented graphically as a "Petri net" concurrent automaton as depicted in Figure 5.1.

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.

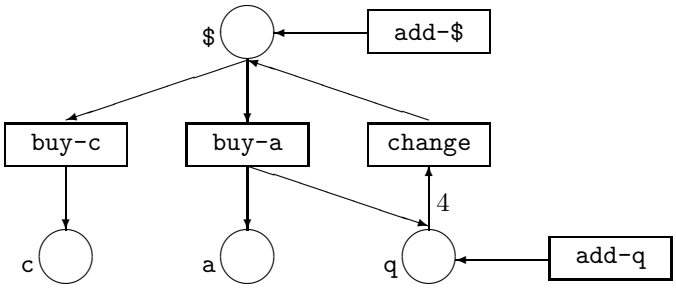


Figure 5.1: Petri net of the vending machine.

The machine is *concurrent*, because we can *push several buttons at once* (that is, 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 planned concurrent implementation of Maude it will be possible to execute concurrently many rewriting steps for a wide range of system modules.

We might have tried a simpler alternative, `null => q`, for 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 thus 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

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

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. Conditions are evaluated from left to right, and therefore the order in which they appear 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. Also, 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 [63]:

```
sorts Label Act Process ActProcess .
subsorts Qid < Label < Act .
subsort Process < ActProcess .

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 illustrated in Section 5.4 and explained in Sections 15.2 and 15.4.

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 module M, a conditional[1] 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

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 $v_i \to w_i$ 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 with 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 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, idempotence, 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; 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* [64] with respect to the equations E modulo A. This means that appropriate "critical pairs" between rules and equations can be joined, allowing us to intermix rewriting with rules and rewriting with equations without losing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken. In this way, we get the effect of rewriting modulo $E \cup A$ with just a matching algorithm for A. In particular, the following method of rewriting a term is then always complete, in the sense that it will not miss any equationally

[1] 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.

equivalent rewrites: first reduce the term to canonical form with the equations E modulo A, and then perform a rewrite step with R. This is exactly what the `rewrite` and `frewrite` commands do at each step (see Section 15.2 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. 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 [8]; 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 [8], 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[2] 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* [45, 8]. 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* [45, 8].

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 $[\alpha] : [t] \rightarrow [t']$, 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 Examples: `rewrite`, `frewrite`, and `search`

Now we illustrate the use of the Maude commands available for system modules. Recall the vending machine example:

```
fmod VENDING-MACHINE-SIGNATURE is
  sorts Coin Item Marking .
  subsorts Coin Item  < Marking .
  op null : -> Marking .
  op __ : Marking Marking -> Marking [assoc comm id: null] .
  op $ : -> Coin [format(r! o)] .
  op q : -> Coin [format(r! o)] .
  op a : -> Item [format(b! o)] .
```

[2] More precisely, each kind k in Σ corresponds to a different choice for a set of states, namely the set $T_{\Sigma/E \cup A,k}$.

```
  op c : -> Item [format(b! o)] .
endfm

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
```

We can use the `rewrite` command (abbreviated `rew`) to explore the behavior of different initial markings. The number between brackets provides an upper bound for the number of rule applications that are allowed.

```
Maude> rew [1] $ $ q q .
rewrite [1] in VENDING-MACHINE : $ $ q q .
rewrites: 1 in 0ms cpu (9ms real) (~ rewrites/second)
result Marking: $ $ q q q

Maude> rew [2] $ $ q q .
rewrite [2] in VENDING-MACHINE : $ $ q q .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Marking: $ $ $ q q q

Maude> rew [3] $ $ q q .
rewrite [3] in VENDING-MACHINE : $ $ q q .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
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) (~ rewrites/second)
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) (~ rewrites/second)
result Marking: $ $ $ $ $

Maude> rew [6] $ $ q q .
rewrite [6] in VENDING-MACHINE : $ $ q q .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result Marking: $ $ $ $ $ q

Maude> rew [200] $ $ q q .
rewrite [200] in VENDING-MACHINE : $ $ q q .
rewrites: 200 in 10ms cpu (10ms real) (20000 rewrites/second)
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-$` until the rule `change` can be applied. The top-down rule-fair `rewrite` strategy keeps trying to apply rules on the top operator (`__` in this case) in a fair way. The rules applicable on the top are `add-q`, `add-$`, 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 left-hand 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] $ $ 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) (~ rewrites/second)
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) (~ rewrites/second)
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) (~ rewrites/second)
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) (~ rewrites/second)
result Marking: $ $ a c
```

The advisory is about the modified rules for buying. It 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 12.2.6 for more details.

Let us see what happens when we use another strategy for rewriting. 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. These rewrite strategies are described in detail in Section 15.2.

```
Maude> frew [2] $ $ q q .
frewrite [2] in VENDING-MACHINE : $ $ q q .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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) (~ rewrites/second)
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 sort of the term. 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`.

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 $ $ .
rewrite in SIMPLE-VENDING-MACHINE : $ $ .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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) (~ rewrites/second)
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) (~ rewrites/second)
result Marking: a c c
```

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 we have 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 f(a, a) .
frewrite in BB-TEST : f(a, a) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Expression: bingo
```

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[3] allows one to explore the reachable state space in different ways.

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. The `=>!` symbol indicates that we are searching for terminal states, that is, states that cannot rewrite further.

```
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) (~ rewrites/second)
M:Marking --> null

No more solutions.
states: 6  rewrites: 5 in 0ms cpu (1ms real) (~ rewrites/second)
```

[3]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 15.

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 only 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] .)

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.

One can also specify a limit to the number of solutions searched for. In the above example there was only one, so a bound would not make any difference. But, returning to the coin generating (and thus nonterminating) vending-machine module `VENDING-MACHINE`, the search space is infinite so it is important to be able to limit the search.

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 .

Solution 1 (state 108)
```

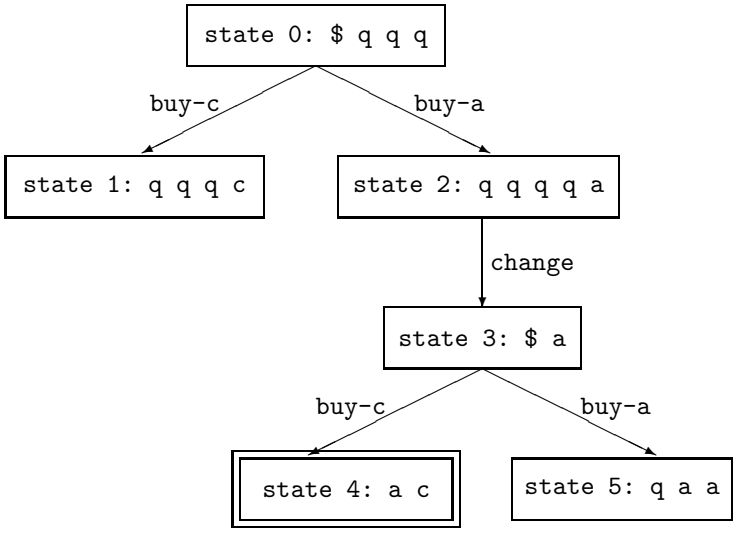


Figure 5.2: Graphical representation of search graph in example.

```
states: 109  rewrites: 1857 in 0ms cpu (41ms real) (~ rewrites/second)
  M:Marking --> q q q q

Maude> cont 1 .
Solution 2 (state 113)
states: 114  rewrites: 185 in 0ms cpu (4ms real) (~ rewrites/second)
M:Marking --> null
```

In case you forget to set a bound on the search or continuation, you can also interrupt a search in progress with 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 12.1.3 and 15.10 for more information on the debugger.

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 rules of the `BB-TEST` module we get:

```
Maude> show rls .
rl a => b .
rl b => a .
rl f(b, b) => bingo .
```

See Chapter 15 for a complete list of commands.

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 [38] 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 [9], OBJ [38], 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 the summation, renaming, and instantiation operations on 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 [24, 30] 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 module expression. Modules are constructed for each subexpression of a module expression, and so each 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 import. 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 15.11 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 of sorts `Nat` or `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 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' 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 [6] together with an inductive theorem prover for membership equational logic and a Church-Rosser checker such as those described in [16], this can be done for functional modules. Using the fact that initial models of rewrite theories are also models of equational theories [8], 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. For 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 in an appropriate sense.

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 do theorem proving to 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 Some 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 example of a vending machine 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
```

Note 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 configurations

Later, in Section 8.1, devoted to the definition of configurations of objects and messages for object-based programming, there are 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 `Status`, 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 MYCONF is
  ex CONFIGURATION .
  sort MsgBody .
  op msg : Oid Oid MsgBody -> Msg [message] .
  sort Status .
  op st :_ : Status -> Attribute .
  op idle : -> Status .
  op wait4 : Oid Oid MsgBody -> Status .
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 the module `BOOL` is imported implicitly and 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 the summation of module expressions is

`ModuleExpression` + `ModuleExpression`

with `+` associative and commutative.

Summation expressions are flattened before being evaluated so `A + (B + C)` and `(C + B) + A` 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 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 FOO is
  protecting FLOAT + STRING .
  ...
endfm
```

6.2.2 Module renaming

The syntax of the 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 on them 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 [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 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 operator attributes. 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 contents that are 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 A is
  sort Foo .
  op a : -> Foo .
  op f : Foo -> Foo .
endfm
```

```
fmod B is
  including A .
  sort Bar .
  subsort Foo < Bar .
  op f : Bar -> Bar .
endfm

fmod C is
  inc B * (op f : Bar -> Bar to g) .
endfm
```

Here, the operator `f` in the module `A` looks as though it is not affected by the renaming in the module `C`, but because of the subsort declaration `Foo < Bar` in `B`, it should be renamed for consistency. This is handled 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 `B`. Thus, the module expression `B * (op f : Bar -> Bar to g)` evaluates to a new module `B * (op f : [Foo,Bar] -> [Foo,Bar] to g)`, which includes the module expression `A * (op f : [Foo,Bar] -> [Foo,Bar] to g)`, which evaluates to a new module `A * (op f : [Foo] -> [Foo] to g)`, in which `f` has been renamed.

In general, `*` does not distribute over `+`. Consider this other example:

```
fmod A is
  sorts Foo Bar .
endfm

fmod B is
  inc A .
  op f : Foo -> Foo .
endfm

fmod C is
  inc A .
  subsort Foo < Bar .
  op f : Bar -> Bar .
endfm
```

It is *not* the case that the module expressions `(B + C) * (op f : Bar -> Bar to g)` and `(B * (op f : Bar -> Bar to g)) + (C * (op f : Bar -> Bar to g))` evaluate to the same module because in the latter the operator `f` occurring in `B` will not be renamed.

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.

6.3 Parameterized programming

Theories, parameterized modules, and views are the basic building blocks of parameterized programming [9, 38]. 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* from the formal interface theory to the corresponding actual module. That is, views provide the interpretation of the actual parameters.

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. 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 have rules as well. 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* interpretation, 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 controled 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. 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 theory `TRIV` (see Section 6.3.3). The theory `TRIV` is predefined in Maude (see Section 7.10.1).[1]

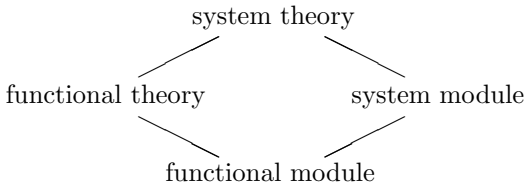
But we can define more interesting theories. For example, the theory of monoids, with an associative binary operator with identity element `1`, can be specified as follows:

[1] Maude's prelude also includes several useful views from `TRIV` to other predefined modules and theories (see Section 7.10.1).

```
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 theory `MONOID`. 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. Theories do not have automatic imports 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 a module may only import a module of less 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. 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.[2]

Let us see a few examples illustrating all this.

The theory of commutative monoids can be defined just as the theory of monoids, but the `_+_` 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
```

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
```

[2] If a theory is imported using a mode other than `including` the system gives an error message that says that the mode is being treated as if it were `including`. Other illegal importations give an error message that says that they are being ignored.

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 theory including all the declarations. The theory of rings can for example be defined 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
```

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 `POSET` 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. The theory of partially ordered sets with an antireflexive and transitive binary operator can be expressed in the following way.

```
fth POSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X < X = false .
  ceq X < Z = true if X < Y and Y < Z [nonexec] .
endfth
```

The theory of totally ordered sets, that is, posets in which all pairs of distinct elements have to be related, can be given as follows:

```
fth TOSET is
  including POSET .
  vars X Y : Elt .
  eq X < Y or Y < X or X == Y = true [nonexec] .
endfth
```

The requirement ensuring the initial semantics of `BOOL` is then preserved by `TOSET` when `POSET` is included. In fact, we are dealing with a structure in which part of it, not only the top theory, has a loose semantics, while other parts contain modules with an initial semantics.

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 [16]. *Such proof obligations are not discharged or checked by the system.*

In the definition of a view we have to indicate its name, 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 expressions, 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 to 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 `:` $S_1 \ldots S_n$ `->` T in the source theory there must exist an operator f' `:` $S'_1 \ldots S'_n$ `->` 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 7.7.

```
view StringAsToset from TOSET to STRING is
  sort Elt to String .
endv
```

Notice that the identity map `op _<_ to _<_` has been omitted.

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 7.5. Notice that we have followed the convention of omitting the "obvious" parts of the map concerning the operators `_+_` and `_*_`, and notice also the use of an operator map sending the operator `e` to the term `1`.

```
view RingToRat from RING to RAT is
  sort Ring to Rat .
  op e to term 1 .
  op z to 0 .
endv
```

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. 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 `s_^1(0)` (see Sections 4.4.2 and 7.2 for details).

As another example, consider the case in which we want to define a view from a theory in which we have a sort `Elt` and a "less or equal" operator `_<=_ : Elt Elt -> Bool`, defined with reflexivity, symmetry and transitivity equations, to a module defining the integers with no such operator but instead with an 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 .
```

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).

[3] In Full Maude (see Chapter 13, maps for all sorts in the source theory have to be given, even when they are identity maps.

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,

```
var X : S .
```

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 `TOSET` to the predefined functional module `INT` of integers (see Section 7.4) in such a way that the `_<_` relation of a toset 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 IntAsToset from TOSET 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 as

```
view IntAsToset from TOSET 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 `TOSET` should hold in the target. Given variables `X` and `Y` 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 and Y <= Z and Y =/= Z .
```

We recommend following the convention of naming views from `TRIV` by the name of the sort to which `Elt` is mapped. Thus, a view from the theory `TRIV` to the module `INT` that sends the sort `Elt` to `Int` should be named `Int`.[4]

```
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.

[4] As we shall see in Section 7.10.1, the view `Int` is predefined in Maude.

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 TOSET from TRIV 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 TOSET from TRIV to TOSET is
endv
```

In this case the view defines an inclusion of theories. In those cases in which the view defines a theory inclusion, we recommend following the convention of naming the view with the name of the target theory.

Let us finish this section by commenting 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.

- `assoc` *operators.* Nested occurrences of associative operators may have been flattened, or have been entered in a flattened way (say `f(a, a, b, b)` for example). 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 module has the 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 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 module X_i `::` T_i is created and included into M, the renamed module X_i `::` T' will also be created and included into X_i `::` T_i.[5]

For example, a parameterized module `SIMPLE-SET` with `TRIV` as interface can be defined as follows:

```
fmod SIMPLE-SET{X :: TRIV} is
  sorts Set NeSet .
  subsorts X$Elt < NeSet < Set .
  op empty : -> Set .
  op _,_ : Set Set -> Set [assoc comm id: empty] .
  op _,_ : NeSet Set -> NeSet [ditto] .
  op _in_ : X$Elt Set -> Bool .
  var E : X$Elt .
  var S : Set .x
  eq E, E = E .
  eq E in E, S = true .
  eq E in S = false [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, one may very easily need in a module sets of integers and sets of quoted identifiers, in which case, given the specification of the `SIMPLE-SET` module above, we would get two `Set` sorts from different importations which would be confused. 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

[5] These renamed modules are visible as names when using the `show modules` command (see Section 15.8) and will be shared, but they cannot be referred to directly in module expressions.

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 `SIMPLE-SET` specification is perfectly valid.

In general, sorts declared in the parameterized module $M\{X_1 :: T_1 , \ldots , X_n :: T_n\}$ may 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 module to define sets could instead have been specified as follows:

```
fmod SET{X :: TRIV} is
  sorts Set{X} NeSet{X} .
  subsorts X$Elt < NeSet{X} < Set{X} .
  op empty : -> Set{X} .
  op _,_ : Set{X} Set{X} -> Set{X} [assoc comm id: empty] .
  op _,_ : NeSet{X} Set{X} -> NeSet{X} [ditto] .
  op _in_ : X$Elt Set{X} -> Bool .
  var E : X$Elt .
  var S : Set{X} .
  eq E, E = E .
  eq E in E, S = true .
  eq E in S = false [owise] .
endfm
```

When this module is instantiated with the view `Int`, the sort `Set{X}` becomes `Set{Int}`, and when it is instantiated with the view `Qid` (see Section 7.10.1) 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]

The above `SET` modules 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`. 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 $Z\$S$. All occurrences of these sorts 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} .
```

[6] In Section 13.3.2, we shall see how this naming convention can be easily extended to the case of Full Maude's parameterized views.

```
  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 .
  eq 2nd(< A ; B >) = B .
endfm
```

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 ordered sets.

```
fmod TOSET-PAIR{X :: TOSET, Y :: TOSET} 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) and by ovals the theories (with loose semantics), and by triple arrows the `protecting` and parameter importations, and by single arrows the `including` importations, we can depict the structure of `TOSET-PAIR` as in Figure 6.1, where we have two copies not only of `TOSET` but also of the `POSET` subtheory, but only one copy of the `BOOL` submodule.

The parameter theory of a module can be any module expression. 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
  pr 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, E' S) = occurrences(E, S) [owise] .
  eq occurrences(E, mt) = 0 .
endfm
```

Module instantiation will be explained in the next section, and then we shall see the execution of some examples.

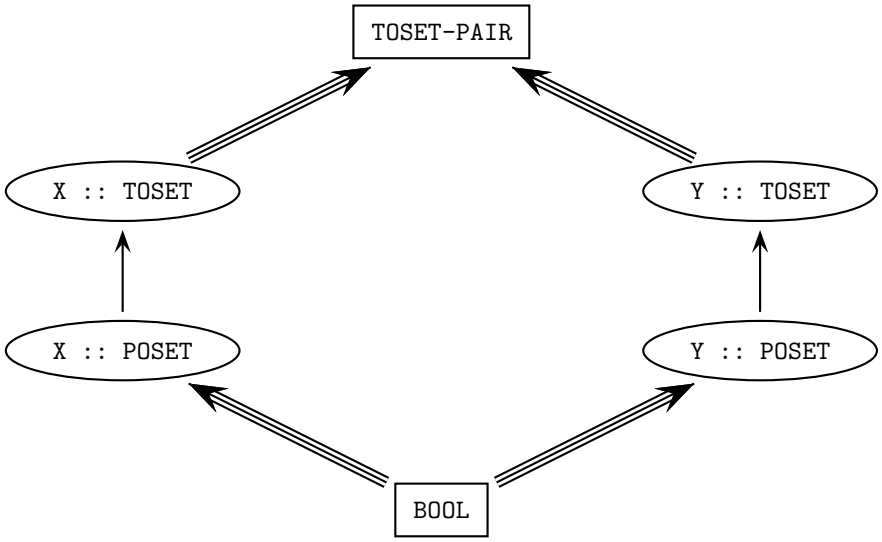


Figure 6.1: Structure of `TOSET-PAIR`.

6.3.4 Module instantiation

Instantiation is the process by which actual parameters are bound to the parameters of a parameterized module or theory 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 target.

The instantiation of a parameterized module has to be made with views explicitly defined previously. Thus, given the views `Int` and `IntAsToset`, introduced in Section 6.3.2, we can have a set of integers with the module expression `SET{Int}`, or pairs of integers as tosets with `TOSET-PAIR{IntAsToset, IntAsToset}`.

As mentioned in Section 6.3.2, we can define views from theories to theories. Using such views we can, for example, instantiate the module `SET` with the view `TOSET` given also in Section 6.3.2. The result is a module `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`, obtaining the module `SET{TOSET}{IntAsToset}`, which defines sets of integers.

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 `SET` and declares the sort `SetList{X}` with constructors `nil` and `_;_`. Note however that `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 have lists of sets of integers, the submodule `SET` also gets instantiated with the same view, providing a definition of sets of integers.[7]

[7] In Section 13.3.2, we shall introduce the notion of *parameterized views*, a more convenient way of defining

```
fmod SET-LIST{X :: TRIV} is
  protecting 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 ring and a set of variables. First, the module `MONOMIAL` defines monomials as terms of the form `X ^ N`, for `X` a variable and `N` the power, with an *empty syntax* multiplication operation on them.

```
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 the monomials, we can specify polynomials as monomials with coefficients in some ring, 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 the theories `RING`, for the coefficients, and `TRIV`, for the variables. Since we need to import the 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$Ring < Poly{R, X} .
  *** multiplication
  op __ : Poly{R, X} Poly{R, X} -> Poly{R, X} [assoc comm] .
  *** addition
  op _++_ : Poly{R, X} Poly{R, X} -> Poly{R, X} [assoc comm] .
  op --_ : Poly{R, X} -> Poly{R, X} .
  op __ : R$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) .
```

this kind of structures. Parameterized views are only available in Full Maude.

```
  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 gets 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 TOSET-PAIR{X :: TOSET, Y :: TOSET} given in Section 6.3.3 using these ideas as follows:

```
fmod TOSET-PAIR{X :: TOSET, Y :: TOSET} is
  protecting PAIR{TOSET, TOSET}{X, Y} .
  op _<_ : Pair{TOSET, TOSET}{X, Y} Pair{TOSET, TOSET}{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 far as there is a (total) order relation available for them. Let us consider the following module MAX, parameterized by the theory TOSET. Given a nonempty 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 (short-circuit disjunction) from the EXT-BOOL module to improve the efficiency of the calculation.

```
fmod MAX{T :: TOSET} is
  protecting 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:

```
fmod INT-MAX is
  protecting 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, given the view StringAsToset introduced in Section 6.3.2 on strings:

```
fmod STRING-MAX is
  protecting 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 this case is the expected one, that is, we get a module or theory parameterized by the targets of those views going to theories.

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.[8]

```
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 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.
- 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.

[8] The view `Qid` is predefined (see Section 7.10.1).

- 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.

Parameterized modules with free parameters cannot be imported: first all of the free parameters must be instantiated away. Parameterized modules with bound parameters only may 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 system module.

Parameterized modules cannot be summed even if all the parameters are bound.

Parameterized modules may be renamed, but the renaming must not touch any sorts or operators coming from a parameter theory. The result of renaming a parameterized module is a parameterized module with the same parameters. Given the `SET-LIST` module above, consider the following example:

```
fmod FOO is
  inc (SET-LIST * (sort Set{X} to MySet{X},
                   op __ : SetList{X} SetList{X} -> SetList{X} to _._)){Qid} .
endfm
```

`SET-LIST` has only a free parameter and so it can be renamed; however its renaming imports the renaming of `SET{X}` which has a bound parameter. 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.[9]

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. Assuming the predefined views `TAO-SET`, from the predefined functional theory `TAO-SET` to `TRIV`, and `Nat<`, from `TAO-SET` to `NAT` (see Section 7.11.6), let us consider the following example:

```
fmod FOO{X :: TRIV} is
  sort Foo{X} .
  op f : Foo{X} -> Foo{X} .
endfm

fmod BAR{X :: TRIV} is
  inc FOO{X} .
  sort Bar{X} .
  op g : Bar{X} -> Foo{X} .
endfm

fmod BAZ is
  inc (BAR * (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')
             ){TAO-SET}{Nat<} .
endfm
```

[9] Currently this is done in a somewhat naive manner and it is possible to trick the implementation by using structured sort names that contain things other than parameters.

In this case, the module FOO gets instantiated before it is renamed:

```
FOO{TAO-SET}{Nat<}
  * (sort Foo{TAO-SET}{Nat<} to Foo'{TAO-SET}{Nat<},
     op f : [Foo{TAO-SET}{Nat<}] -> [Foo{TAO-SET}{Nat<}] to f')
```

Passing parameters from an enclosing module in nonfinal instantiations is prohibited. This restriction avoids many subtle issues. Thus:

```
fmod TEST{X :: RING, Y :: POSET} is
  inc POLYNOMIAL{X, POSET}{Y} .
endfm
```

is *illegal* because X occurs in the nonfinal instantiation POLYNOMIAL{X, POSET}. Given the view POSET from TRIV to POSET introduced in Section 6.3.2, this example can be written as

```
view RING from RING to RING is
endv

view POSET from TRIV to POSET is
endv

fmod TEST{X :: RING, Y :: POSET} is
  inc POLYNOMIAL{RING, POSET}{X, Y} .
endfm
```

Another way of viewing the 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 A specification of sorted lists

In this section we present a specification of sorted lists, which are defined as lists in which their elements are sorted, that is, we define sorted lists as a subsort of lists. To be able to declare memberships defining such data structure, the list concatenation operator is declared at the kind level—membership axioms can only be given on associative operators defined at the kind level (see Section 12.2.8). We then add operations length, min, and max, returning, respectively, the length of a list, and the smallest and greatest element of a sorted list. We assume the theory POSET defined in Section 6.3.1, which is used as parameter of the module SORTED-LIST below, providing a partial order on the elements of the list.

```
fmod LIST-KIND{X :: TRIV} is
  pr NAT .
  sorts NeKList{X} KList{X} .
  subsort X$Elt < NeKList{X} < KList{X} .
  op nil : -> KList{X} .
  op __ : KList{X} KList{X} ~> KList{X} [assoc id: nil] .
  mb NL:NeKList{X} NL':NeKList{X} : NeKList{X} .

  op length : KList{X} -> Nat .
  eq length(N:X$Elt L:KList{X}) = 1 + length(L:KList{X}) .
  eq length(nil) = 0 .
endfm
```

```
fmod SORTED-LIST{X :: POSET} is
  pr LIST-KIND{POSET}{X} .
  sorts NeSortedList{X} SortedList{X} .
  subsort X$Elt < NeSortedList{X}
                < NeKList{POSET}{X} SortedList{X}
                < KList{POSET}{X} .

  vars N M : X$Elt .
  var  SL : SortedList{X} .
  var  L : KList{POSET}{X} .

  op nil : -> SortedList{X} .
  cmb N M L : NeSortedList{X} if N < M /\ M L : SortedList{X} .

  op min : NeSortedList{X} -> X$Elt .
  ceq min(N L) = N if N L : SortedList{X} .

  op max : NeSortedList{X} -> X$Elt .
  ceq max(L N) = N if L N : SortedList{X} .
endfm
```

The following reductions illustrate the behavior of the specification.

```
view NatAsPoset from POSET to NAT is
  sort Elt to Nat .
endv

fmod NAT-SORTED-LIST is
  pr SORTED-LIST{NatAsPoset} .
endfm

Maude> red in NAT-SORTED-LIST : length(2 3 4 5 6 7) .
result NzNat : 6
```

The operations `min` and `max` over a sorted list work as expected.

```
Maude> red in NAT-SORTED-LIST : min(2 3 4 5 6 7) .
result NzNat : 2

Maude> red in NAT-SORTED-LIST : max(2 3 4 5 6 7) .
result NzNat : 7
```

The same operations cannot be applied to a non-sorted list, returning a non-reduced term in the corresponding kind.

```
Maude> red in NAT-SORTED-LIST : min(2 4 5 6 3 7) .
result [KList{POSET}{NatAsPoset}]: min(2 4 5 6 3 7)

Maude> red in NAT-SORTED-LIST : max(2 4 5 6 3 7) .
result [KList{POSET}{NatAsPoset}]: max(2 4 5 6 3 7)
```

We can also have sorted lists of other data elements, for instance of strings:

```
view StringAsPoset from POSET to STRING is
  sort Elt to String .
endv
```

To avoid the confusion between the `length` operator on strings and the one on lists, we rename the module `SORTED-LIST` before instantiating it.

```
fmod STRING-SORTED-LIST is
  pr (SORTED-LIST
        * (op length : KList{POSET}{X} KList{POSET}{X} -> KList{POSET}{X}
               to klength))
        {StringAsPoset} .
endfm

Maude> red in STRING-SORTED-LIST : "one" "two" "three" .
result NeKList{POSET}{StringAsPoset}: "one" "two" "three"

Maude> red in STRING-SORTED-LIST : min("one" "two" "three") .
result [KList{POSET}{StringAsPoset}]: min("one" "two" "three")

Maude> red in STRING-SORTED-LIST : max("one" "three" "two") .
result String: "two"
```

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 `protecting` 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, and the module `BOOL` is imported implicitly.

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 10, the `LOOP-MODE` module in Section 11.1, and the `CONFIGURATION` module is discussed in Section 8.1.

Many operators in predefined modules have the attribute `special` 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 to appropriate C++ code by "hooks" which are specified following the `special` attribute identifier. 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; 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.

7.1 Boolean values

There are five predefined modules involving Boolean values, namely, `TRUTH-VALUE`, `TRUTH`, `BOOL`, `EXT-BOOL`, and `IDENTICAL`. The most basic one is `TRUTH-VALUE`, which has the following definition.

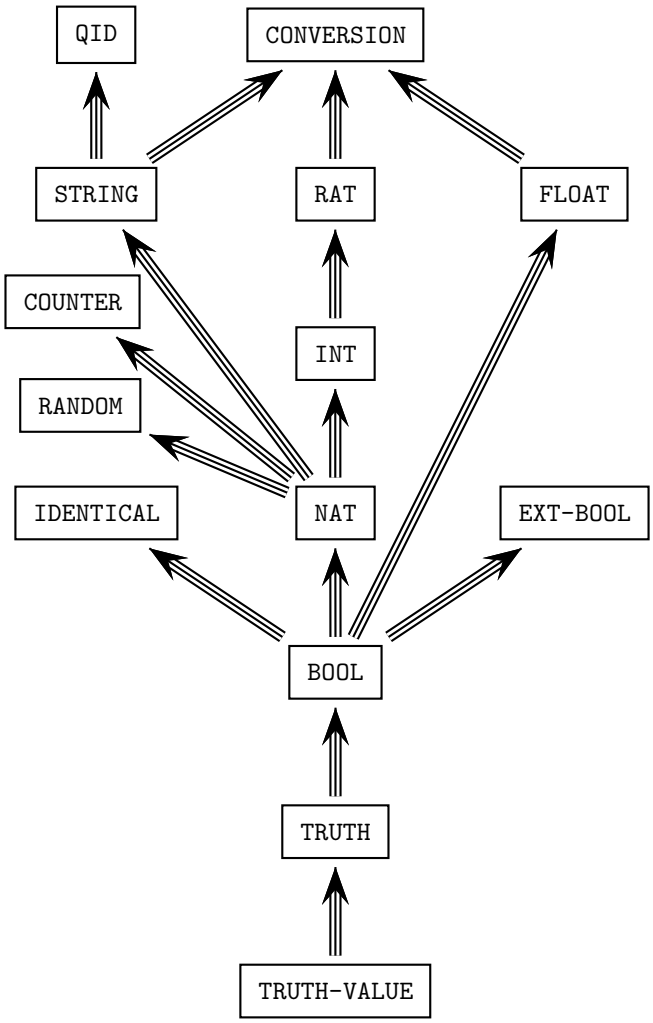


Figure 7.1: Importation (protecting) graph of predefined modules.

```
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 to 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 to 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 operator `if_then_else_fi`, the attribute `poly (2 3 0)` means that `if_then_else_fi` is polymorpic in its second and third arguments as well as in its result.

The operator `if_then_else_fi` 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

[1] The `prec` attribute in the last two operators assigns each of them an appropriate precedence value for parsing purposes (see Section 3.9).

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 like the equality operators since they are part of the language. 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 terminating 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, and membership in a sort are also computable functions. Therefore, by a well-known theorem of Bergstra and Tucker [3], 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` imports `TRUTH` and adds the usual conjunction, disjunction, exclusive or, negation, and implication operators.[2] These operators are defined entirely equationally.

```
fmod BOOL is
  protecting TRUTH .
  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
```

[2] See Section 3.9 for information on precedence values and gathering patterns.

As noted above the BOOL module is imported (in protecting mode) by default as a submodule of any other module defined by the user. This is accomplished by the command

```
set protect BOOL on .
```

that appears in the standard library file prelude.maude. The set protect 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, he/she could write

```
set protect BOOL off .
set protect TRUTH on .
```

Similar commands are available for enabling and disabling implicit importation in extending and including modes. For example, the BOOL module can be extended instead of protected by writing

```
set protect BOOL off .
set extend 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.[3] 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
  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
```

The last module involving truth values is the IDENTICAL module. Like EXT-BOOL, it is not included by default in other modules. That is, it has to be imported explicitly if it is needed. When imported into a module, it adds to each of its connected components polymorphic operators for *syntactic* equality and inequality; that is, two ground terms are compared for syntactic equality—modulo the equational axioms in the attributes of the operators in the module—without performing any reduction of the terms by the equations in the module.

```
fmod IDENTICAL is
  op _===_ : Universal Universal -> Bool
    [prec 51 poly (1 2) strat (0) special ( ... )] .
  op _=/==_ : Universal Universal -> Bool
    [prec 51 poly (1 2) strat (0) special ( ... )] .
endfm
```

[3]Note that EXT-BOOL, as IDENTICAL below and any other module after the predefined module BOOL is set to be "protected" by default, imports (in protecting mode) BOOL, and therefore such importation is not made explicit in the module.

To illustrate this, we combine the NUMBERS module of Chapter 4 and the IDENTICAL module in

```
fmod TEST is
  pr NUMBERS .
  pr IDENTICAL .
endfm
```

We have the following reductions in TEST:

```
Maude> red in TEST : zero + zero == zero .
result Bool: true

Maude> red zero + zero === zero .
result Bool: false

Maude> red zero + zero =/== zero .
result Bool: true

Maude> red zero + s zero === s zero + zero .
result Bool: true
```

When the module BOOL is imported, the system automatically adds to the module an operator to test for membership, `_:: S`, for each sort S (see Section 3.9.3). Analogously to the operators for syntactic equality, the system also adds an operator to test for syntactic membership, `_::: S`, for each sort S. Like the predicates `_===_` and `_=/==_`, the predicate `_::: S` is a syntactic check, using only sort information and the equational axioms in the attributes of the operators in the module, without any rewriting using the equations in the module to test for membership.

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) ::: Zero .
result Bool: false

Maude> red sd(zero, zero) ::: Nat .
result Bool: true

Maude> red (zero nil) ::: Zero .
result Bool: true
```

The syntactic membership `sd(zero, zero) ::: Zero` rewrites to `false` because user-defined equations are needed for Maude to rewrite `sd(zero, zero)` to `zero`. In contrast, `(zero nil)` has sort NatSeq 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 include 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`) by the successor operator `s_`.

```
fmod NAT is
  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 sucessor notation; for example:

```
fmod FACT is
  inc 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 OPERATORS
  *** 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 `((i quo j) * j) + (i rem j)` = `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.

```
   *** 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, negation, 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 (for more information, see `http://www-personal.engin.umich.edu/~wagnerr/MersenneTwister.html`). 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 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 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 15.2), as well as the `erewrite` command (see later Section 8.4), 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.

```
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) => if rand < P:Float then true else false fi .
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 .
  op broken : Nat Float -> Clock .
  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
```

The 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 15.2). 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) .
rewrite in CLOCK : clock(0, 1.0e+3) .
rewrites: 489 in 10ms cpu (19ms real) (48900 rewrites/second)
result Clock: broken(40, 9.607702107358117e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
rewrite in CLOCK : clock(0, 1.0e+3) .
rewrites: 554 in 0ms cpu (3ms real) (~ rewrites/second)
result Clock: broken(46, 9.5501998182355942e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
rewrite in CLOCK : clock(0, 1.0e+3) .
rewrites: 215 in 0ms cpu (1ms real) (~ rewrites/second)
result Clock: broken(16, 9.8411944181564002e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
rewrite in CLOCK : clock(0, 1.0e+3) .
rewrites: 101 in 0ms cpu (0ms real) (~ rewrites/second)
result Clock: broken(6, 9.9401498001499397e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
rewrite in CLOCK : clock(0, 1.0e+3) .
rewrites: 348 in 0ms cpu (2ms real) (~ rewrites/second)
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 inert with respect to search, model checking, and equational rewriting. Notice that there are potentially bad interactions with the debugger (see Section 12.1.3) 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
*** negation
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
  inc 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 cond-eqns). However, memberships aren't guaranteed to work correctly with the number hierarchy because of the special internal representation for iterated towers of `s_` symbols.

7.5 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 have 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
```

7.6 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
  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 `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
```

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
```

7.7 Strings

The module `STRING` declares sorts and operators for manipulating strings. Strings of length one form a subsort `Char` of `String`. Operations on strings are based on the SGI rope package [4], 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 [39, 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.

```
*** 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 ( ... )] .
```

Remember that string indexing begins with 0. `substr(S:String, Start:Nat, Len:Nat)` returns the substring of `S:String` of length `Len:Nat` beginning at position `Start:Nat`. If `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 equations 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`.

```
let
  length(S) = K
  length(P) = J
in
  I <= find(S, P, I) <= K - J
  0 <= rfind(S, P, I) <= min(I, K - J)

  find(S, S, 0) = 0 = rfind(S, S, I)
  find(S, "", I) = if (I <= K) then I else notFound fi
  rfind(S, "", I) = if (I >= K) then K else I fi

  find(S, P, I) =/= notFound implies
    substr(S, 0, find(S, P, I)) + P + substr(S, find(S, P, I) + J, K) = S

  rfind(S, P, I) =/= 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
```

7.8 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

```
float(rat(F:FiniteFloat)) = F:FiniteFloat
```

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, integer or 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 1.1 as 1.1000000000000001, using 17 significant digits. The above representation is obtained by reducing `decFloat(1.1, 52)`.)

7.9 Quoted identifiers

The module `QID` is a wrapper for strings in order to provide a Maude representation for tokens of Maude syntax. Quoted identifers 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 ( ... )] .

  ***  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
```

[4] The syntax of Maude identifiers is discussed in Section 3.1.

The operator `qid` is only injective on strings without white space, control characters, and certain other characters which are converted to backquote. Thus `qid(string(q)) = q` 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`.

7.10 Basic theories and standard views

The library of predefined modules provided by Maude in the file `prelude.maude` 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.

7.10.1 TRIV

As already described in Section 6.3.1, the simplest nonempty theory is called `TRIV` and consists of a single sort. The intuition behind this simple theory is that the minimum requirement possible on a parameterized data type construction is having a data type to build more data on top of it.

```
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 as 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

view String from TRIV to STRING is
  sort Elt to String .
endv

view Qid from TRIV to QID is
  sort Elt to Qid .
endv
```

7.10.2 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
```

There is a view from `TRIV` to `DEFAULT` that forgets the "default" element. The name of this view 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
```

7.10.3 TAO-SET

The predefined theory `TAO-SET` describes a set with a *transitive and antisymmetric order*, that we abbreviate *TAO*. This notion of order on a set generalizes (i.e., it is weaker than) the notions of partial orders and strict partial orders. In order to make it easier for the reader to compare these notions, we review them and two more here:

- A *preorder* (or quasi-order) is a binary relation that is reflexive and transitive.
- A *partial order* is a binary relation that is reflexive, transitive, and antisymmetric.
- An *equivalence relation* is a binary relation that is reflexive, transitive, and symmetric.
- A *strict partial order* is a binary relation that is irreflexive and transitive (which imply antisymmetry).

The notion of a TAO is used below, in Section 7.11.6, to define sortable lists, where the properties of reflexivity/irreflexivity are not necessary.

Notice that `TAO-SET`, as presented below, imports the theory `TRIV` and also (in `protecting` mode) the module `BOOL`. The two conditional equations express in a non-executable way (due to the occurrence of new variables in the condition of each equation) the two required properties of the binary relation `_<_` on the sort `Elt`, as made explicit in the corresponding labels.

```
fth TAO-SET is
  protecting BOOL .
  including TRIV .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  ceq X < Z = true if X < Y and Y < Z [nonexec label transitive] .
  ceq X = Y if X < Y /\ Y < X [nonexec label antisymmetric] .
endfth
```

There is a view from `TRIV` to `TAO-SET` that forgets the order (and its properties). The name of this view coincides with the name of the target theory.

```
view TAO-SET from TRIV to TAO-SET is endv
```

The Maude library includes appropriate views that map from `TAO-SET` to several built-in data type modules by selecting the main sort and the standard strict 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 `TAO-SET` into `RAT` is named `Rat<`.

```
view Nat< from TAO-SET to NAT is
  sort Elt to Nat .
endv

view Int< from TAO-SET to INT is
  sort Elt to Int .
endv

view Rat< from TAO-SET to RAT is
  sort Elt to Rat .
endv

view Float< from TAO-SET to FLOAT is
  sort Elt to Float .
endv

view String< from TAO-SET 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.

7.11 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 7.3).

Other container data types may be added in the future.

7.11.1 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 nonempty lists.

```
fmod LIST{X :: TRIV} is
```

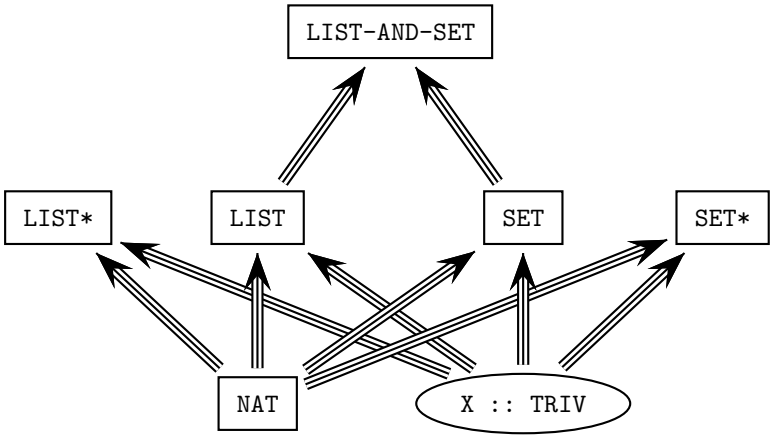


Figure 7.2: Importation graph of parameterized list and set modules.

```
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] .

var 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 10).

```
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
```

7.11.2 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 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 for 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) .
endfm
```

The Maude metalevel (see Chapter 10) 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
```

7.11.3 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 .
  var 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.

Conversely, `makeList` transforms a set into a list. Although the equations make it look as the inverse of the previous one, notice that this process imposes an order into a set of elements; this order is system defined, since it is an order determined by the implementation of associative-commutative rewriting. Moreover, when `makeList` is applied to a set, the possible repetitions of elements in the set representation are also eliminated. Therefore, `makeList` is not a real inverse, but only a partial inverse to `makeSet`; in general, for a set `S` and a list `L` we have `makeSet(makeList(S))` = `S` but instead `makeList(makeSet(L))` $\neq$ `L`.

These are the corresponding definitions by structural induction on lists and sets, respectively.

```
  op makeSet : List{X} -> Set{X} .
  op makeSet : NeList{X} -> NeSet{X} .
  eq makeSet(nil) = empty .
```

```
  eq makeSet(E L) = E, makeSet(L) .

  op makeList : Set{X} -> List{X} .
  op makeList : NeSet{X} -> NeList{X} .
  eq makeList(empty) = nil .
  eq makeList((E, S)) = E makeList(S) .
```

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.

```
  op filter : List{X} Set{X} -> List{X} .
  eq filter(nil, S) = nil .
  eq filter(E L, S) = if E in S then E filter(L, S) else filter(L, S) fi .

  op filterOut : List{X} Set{X} -> List{X} .
  eq filterOut(nil, S) = nil .
  eq filterOut(E L, S) = if E in S then filterOut(L, S) else E filterOut(L, S) fi .
endfm
```

For illustration, we consider the following instantiation:

```
fmod INT-LS is
  pr LIST-AND-SET{Int} .
endfm

Maude> red in INT-LS : 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
```

Notice that in the following first reduction we get a list different from the original, while in the second reduction we get a different representation of the same set.

```
Maude> red in INT-LS : makeList(makeSet(1 -1 1 -2 1)) .
result NeList{Int}: 1 -1 -2

Maude> red makeSet(makeList((5, 4, 3, 4, 5))) .
result NeSet{Int}: 3, 4, 5
```

7.11.4 Generalized lists

With the construction of parameterized lists described in Section 7.11.1, 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 7.11.1, 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} .
  var 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 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 we have seen in Section 7.11.1, 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
```

7.11.5 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."

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 theory `TRIV` in the form `X$Elt`.

The module `SET*` provides for generalized sets the same operations we have seen in Section 7.11.2 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 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. As in Section 7.11.2, the operators `delete` and `_in_` are defined by means of the `otherwise` attribute. Moreover, each one has an additional equation for the singleton case, that 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 7.11.2, 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)) .
endfm
```

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}}
```

7.11.6 Sortable lists

Given an asymmetric and transitive order (TAO) $<$ on a set S of elements, we will say that a list L over S is *sorted* if for every pair (u, v) of members in L with u occurring before v and $u \neq v$, it is the case that $v < u$ is false. With this definition, neither reflexivity or irreflexivity is needed, and therefore the parameterized construction of sortable lists is precisely based on the requirements provided by the theory `TAO-SET` (see Section 7.10.3).

This specification of sortable lists imports "standard" lists (from Section 7.11.1), but first it is necessary to match the parameter `TRIV` of lists with the parameter `TAO-SET` of sortable lists. This is accomplished by means of the predefined view `TAO-SET` from `TRIV` to `TAO-SET` that forgets the order and its properties (see Section 7.10.3). 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 the renaming has been abbreviated to α, and where the different types of arrows represent the different relationships bewteen modules: importation, views between theories, instantiation, and renaming.

```
fmod SORTABLE-LIST{X :: TAO-SET} is
  protecting (LIST{TAO-SET} *
              (sort NeList{TAO-SET}{X} to NeList{X},
               sort List{TAO-SET}{X} to List{X})){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;

[5] We realize that terminology here can be a bit confusing, because in Maude `sort` is also a keyword for types.

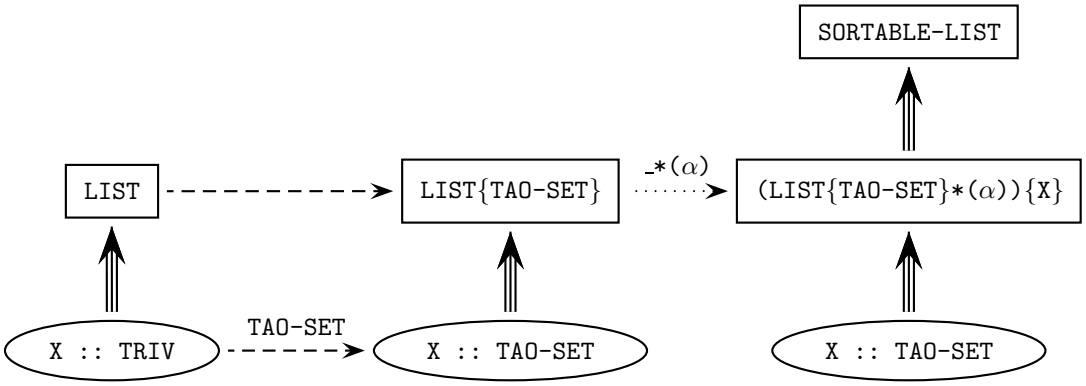


Figure 7.3: From lists to sortable lists.

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') .
```

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' == true then $merge(L, E' L', A E)
      else $merge(E L, L', A E')
      fi .
endfm
```

We take the predefined view `String<` from `TAO-SET` to `String` (where `<` is the lexicographic order on strings) and instantiate the previous module before doing some sample reductions.

```
fmod STRING-SORTING is
  pr SORTABLE-LIST{String<} .
endfm

Maude> red in STRING-SORTING :
      $split("a" "quick" "brown" "fox" "jumps" "over" "the" "lazy" "dog", nil, nil) .
result $Split{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"
```

7.12 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 a set of pairs like the previous one are different. This situation 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.

7.12.1 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
  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.

```
  op insert : X$Elt Y$Elt Map{X,Y} -> Map{X,Y} .
  eq insert(D, R, (M, D |-> R')) = (M, D |-> R) .
  eq insert(D, R, M) = (M, D |-> R) [owise] .
```

The look up operator is represented with the notation `_[_]`. Again, matching and `owise` are used to distinguish whether the second argument appears in the domain of definition of the map provided as first argument or not. When the answer is afirmative, the result is the value provided in the corresponding entry; when the answer is negative, the result is the constant `undefined` in the kind, with the self-explanatory meaning that the map is undefined on the given argument.

```
  op _[_] : Map{X,Y} X$Elt -> [Y$Elt] .
  eq (M, D |-> R)[D] = R .
  eq M[D] = undefined [owise] .
endfm
```

We use the predefined views `String` and `Nat` (see Section 7.10.1) 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
```

The following reductions show that the undesired repetition of a domain value in two entries of the same map may have unexpected results.

```
Maude> red in STRING-NAT-MAP : ("a" |-> 3, "a" |-> 4)["a"] .
result NzNat: 3

Maude> red ("a" |-> 3, "a" |-> 2)["a"] .
result NzNat: 2
```

7.12.2 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 7.10.2).

The constructor for entries is named `_|->_`, as for maps, while the set constructor is denoted here `_;_`.

```
fmod ARRAY{X :: TRIV, Y :: DEFAULT} is
  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.

```
  op insert : X$Elt Y$Elt Array{X,Y} -> Array{X,Y} .
  eq insert(D, R, (A ; D |-> R')) = if R == 0 then A else (A ; D |-> R) fi .
  eq insert(D, R, A) = if R == 0 then A else (A ; D |-> R) fi [owise] .
```

The definition of the look up operator for arrays only differs from the one for maps in the occurrence of the default value `0` instead of the constant `undefined`.

```
  op _[_] : Array{X,Y} X$Elt -> [Y$Elt] .
  eq (A ; D |-> R)[D] = R .
  eq A[D] = 0 [owise] .
endfm
```

We do the same instantiation for arrays as for maps, with the predefined views `String` from Section 7.10.1 and `Nat0` from Section 7.10.2).

```
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
```

7.13 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 equations in m variables having the form `M v = 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 7.12.2.

First it is created a data type of pairs of natural numbers as indices for matrices.

```
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 7.10.2

```
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 7.10.1. 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 =/= 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 nonempty and `B` is empty), or infinitely many solutions (in this case `A` and `B` are both nonempty).

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 [59]). 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 [21] 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 [58],

$$\begin{pmatrix} 1 & 2 & -1 & 0 & -2 & -1 \\ 0 & -1 & -2 & 2 & 0 & 1 \\ 2 & 0 & 1 & -1 & -2 & 0 \end{pmatrix} \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

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 rewrites/second)
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 [53], 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 [21]:

```
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") .
rewrites: 1 in 0ms cpu (2ms real) (~ rewrites/second)
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
```

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. This is explained in detail below.

As discussed in Chapter 14, 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` 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* and 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 use of 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 14). 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).

```
  *** 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 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 stick to 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 : 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, `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 messages `debit(A-003, 300)` and `credit(A-002, 300)` 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] .
var N' : Nat .
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 `debit(A-001, 150)` is processed.

```
Maude> rew bankConf .
rewrite 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 `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 Section 15.4).

```
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) (~ rewrites/second)
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) (~ rewrites/second)
```

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: `new-account(A-000, A-004, 100)`. Of course, in a real system more care would be needed to assure unique account identities, etc. 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 `mgrConf`:

```
Maude> rew mgrConf .
rewrite in BANK-MANAGER-TEST : mgrConf .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
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 is 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. As an example we model a *Ticker*, the classic example of an actor [1, 57]. 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

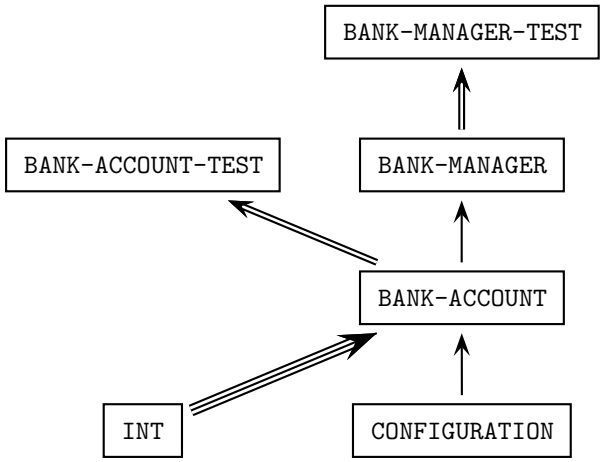


Figure 8.1: Importation graph of bank modules.

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. The ticker sends the current value of its counter in response to a `time-req` message. `Ticker(T:Aid, N:Nat)` is an actor with identifier `T:Aid` and counter value `N:Nat`.

```
mod TICKER is
  inc ACTOR-CONF .
  protecting NAT .
  op Ticker : Aid Nat -> Actor [ctor] .
  op tick : -> MsgBody [ctor] .
  op time-req : Aid -> MsgBody [ctor] .
  op time-reply : 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 <| time-req(C))
    => Ticker(T, N) (C <| time-reply(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 `time-req` message from `me`.

```
mod TICKER-TEST is
  ex TICKER .
  ops myticker me : -> Aid [ctor] .
  op tConf : -> AConf .
  eq tConf
    = Ticker(myticker, 0)
      (myticker <| tick)
      (myticker <| time-req(me)) .
endm
```

If we ask Maude to rewrite `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, ten rewrites.

```
Maude> rew [10] tConf .
rewrite [10] in TICKER-TEST : tConf myticker <| time-req(me) .
rewrites: 11 in 0ms cpu (0ms real) (~ rewrites/second)
result AConf: (myticker <| tick) (me <| time-reply(1)) Ticker(myticker, 9)
```

We see that the `time-req` message was processed after just one `tick` was processed.

An interesting property of this configuration is that the reply to the `time-req` message can contain an arbitrarily large natural number, since any number of `tick`s could be processed before the `time-req`. For particular numbers this can be checked using the search command.

```
Maude> search [1] tConf =>+ tc:AConf me <| time-reply(100) .
search [1] in TICKER-TEST : tConf =>+ tc:AConf me <| time-reply(100) .

Solution 1 (state 5152)
states: 5153  rewrites: 5153 in 0ms cpu (285ms real) (~ rewrites/second)

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`.

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] .
  ops 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 <| time-req(C) .

  rl [time-reply] :
    Cust2(C, TF, N) (C <| time-reply(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.

```
Maude> rew [40] ic2 .
rewrite [40] in TICKER-FACTORY-TEST : ic .
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)
```

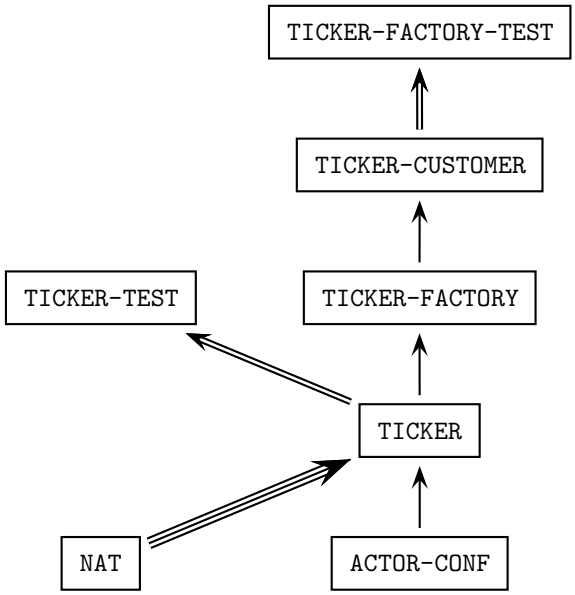


Figure 8.2: Importation graph of ticker modules.

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) <| time-req(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 12.1.4) 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. To enable the object-message fair rewriting we need only do three things:

- give actor configurations the `config` attribute,
- give the message constructor the `message` attribute, and
- give each actor 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) .
frewrite [4] in O-TICKER-FACTORY-TEST : ic2 Cust(c1, tf, 0) .
Warning: saw duplicate object: Cust1(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 behavior 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. 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. There may be multiple

configuration, object and message constructors. A rule is considered to be an object-message rule if the following requirements hold:

(a) Its lefthand side has a configuration constructor on top with two arguments A and B,

(b) A and B are stable (that is, they cannot change their top symbol under a substitution),

(c) A has a message constructor on top,

(d) B has an object constructor on top, and

(e) the first arguments of A and B are identical.

For example in the module `O-TICKER-CUSTOMER` the rules labeled `newReply` and `time-reply` 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 with the descent function `metaFrewrite`—see Section 10.4.2). When the fair traversal attempts to perform a single rewrite on a term headed by a configuration constructor, the following happens:

(a) Arguments headed by object constructors are collected. It is a runtime error for more than one object to have the same name.

(b) For each object, messages with its name as first argument are collected and placed in a queue.

(c) Any remaining arguments are placed on a remainder list.

(d) 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.

(e) 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

[1] Assuming, as it should be the case, that both object constructors have been declared with the `object` attribute.

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 there are just two data agents in the system, and that once the value of a key is set, globally, it will not be changed.

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 message body, and are constructed with the `msg` constructor. The technique for maintaining task information (we assume at most one task per object at any given time) is to define a sort `Status` and a status attribute, `st`, that records the object's task status. (This would be called the continuation in the functional programming community.) The status constant `idle` indicates that an object has no active task. The status `wait4(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`. Additional status constructors can be defined if needed.

The module `MYCONF` extends `CONFIGURATION` with the uniform message syntax and status attribute.

```
mod MYCONF is
  ex CONFIGURATION .

  *** my msg syntax
  sort MsgBody .
  op msg : Oid Oid MsgBody -> Msg [ctor message] .

  *** implement blocking by a wait4 attribute
  sort Status .
  op st :_ : Status -> Attribute [ctor] .
  op idle : -> Status [ctor] .
  op wait4 : Oid Oid MsgBody -> Status [ctor] .
endm
```

[2] There are transformations that internalize conditions on message delivery and to ensure a stronger fairness condition [44].

A data agent stores a dictionary mapping keys to data elements. The module `DICT` defines the dictionary data type `Dict`, with lookup and updating operations. For simplicity we take both keys and data elements to be quoted identifiers, and we use `'nothing` to indicate that there is no entry.

```
fmod DICT is
  pr QID .
  sorts Row Dict .
  subsort Row < Dict .
  op none : -> Dict [ctor] .
  op __ : Dict Dict -> Dict [ctor assoc comm id: none] .
  op r : Qid Qid -> Row [ctor] .

  op lookup : Dict Qid -> Qid .
  op update : Dict Qid Qid -> Dict .

  vars Key Q : Qid .
  vars Val New : Qid .
  var  D : Dict .

  eq lookup(r(Key, Val) D, Key) = Val .
  eq lookup(D, Key) = 'nothing [owise] .

  eq update(r(Key, Val) D, Key, New) = r(Key, New) D .
  eq update(D, Key, New) = r(Key, New) D [owise] .
endfm
```

Note that dictionaries generated from the empty dictionary (`none`) by updating will contain at most one row for a given key. Notice also the use of the `owise` attribute in defining the `lookup` and `update` functions. The standard recursive definitions might look like the following:

```
  eq lookup(none, Key) = 'nothing .
  eq lookup((r(Q, Val) D), Key)
    = if (Q == Key) then Val else lookup(D, Key) fi .

  eq update(none, Key, New) = r(Key, New) .
  eq update((r(Q, Val) D), Key, New)
    = if (Q == Key)
      then (r(Key, New) D)
      else (r(Q, Val) update(D, Key, New))
      fi .
```

Using `owise` is both simpler, and more efficient. (In the sample runs below, the difference is roughly a factor of two.)

We split the specification of data agents into two modules: the `DATA-AGENTS-INTERFACE` functional module, which defines the interface, and the `DATA-AGENTS` system module, 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 the different modes in the importing declarations: `MYCONF` is imported in `extending` mode, because we add data to the old sorts, but without making further identifications (the interface module has no equation), while `DICT` is imported in `protecting` mode, because the dictionary data is only used as an argument, without modifying those sorts at all.

```
fmod DATA-AGENTS-INTERFACE is
  ex MYCONF .
  pr DICT .

  *** interface / messages
  op getReq : Qid -> MsgBody [ctor] .
  op getReply : Qid Qid -> MsgBody [ctor] .
  op tellReq : Qid Qid -> MsgBody [ctor] .
  op tellReply : Qid Qid -> MsgBody [ctor] .
  op lookupReq : Qid -> MsgBody [ctor] .
  op lookupReply : Qid Qid -> MsgBody [ctor] .

  *** class structure
  op DataAgent : -> Cid [ctor] .
  op data :_ : Dict -> Attribute [ctor] .
  op pal :_ : Oid -> Attribute [ctor] .
endfm
```

In a request-reply style of interaction, message body constructors come in pairs. For example, (`getReq`, `getReply`) constitute the message body pair for an agent to request data from a pal and identify the reply. Similarly (`lookupReq`, `lookupReply`) and (`tellReq`, `tellReply`) are the message body pairs used when interacting with a customer who wants to access and set data values.

A data agent has class identifier `DataAgent`. In addition to a status attribute, `st`, 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 `sam`'s initial state might look like

```
< sam : DataAgent | data : none, pal : joe, st : idle >
```

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.)

```
mod DATA-AGENTS is
  inc DATA-AGENTS-INTERFACE .
  vars A O C : Oid .
  var  D : Dict .
  vars Key Val : Qid .
  var  Atts : AttributeSet .
  var  Any : Status .

  rl [lookup] :
    < A : DataAgent | data : D, pal : O, st : idle >
    msg(A, C, lookupReq(Key))
    => if lookup(D, Key) == 'nothing
       then < A : DataAgent | data : D, pal : O, st : wait4(O, C, lookupReq(Key)) >
            msg(O, A, getReq(Key))
       else < A : DataAgent | data : D, pal : O, st : idle >
            msg(C, A, lookupReply(Key, lookup(D, Key)))
       fi .

  *** request missing data from pal
  rl [getReq] :
    < A : DataAgent | data : D, pal : O, Atts >
```

```
    msg(A, C, getReq(Key))
    => < A : DataAgent | data : D, pal : O, Atts >
       msg(C, A, getReply(Key, lookup(D, Key)))  .

  *** receive requested missing data from pal
  rl [getReply] :
    < A : DataAgent | data : D, pal : O, st : wait4(O, C, lookupReq(Key)) >
    msg(A, O, getReply(Key, Val))
    => < A : DataAgent | data : update(D, Key, Val), pal : O, st : idle >
       msg(C, A, lookupReply(Key, Val))  .

  rl [tell] :
    < A : DataAgent | data : D, pal : O, st : Any >
    msg(A, C, tellReq(Key, Val))
    => < A : DataAgent | data : update(D, Key, Val), pal : O, st : Any >
       msg(C, A, tellReply(Key, Val))  .
endm
```

The rule labeled `lookup` specifies how an agent handles a `lookupReq` message. An agent can only receive a `lookupReq` if its status is `idle`. The agent first looks to see if its local dictionary contains the requested entry. If `lookup(D, Key) == 'nothing`, then a `getReq` is sent to the `pal` and the agent waits for a reply, remembering the pending lookup request (`st : wait4(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 `'nothing`, or 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 waiting status (`st : wait4(O, C, lookupReq(Key))`. The agent records the received reply (`data : update(D, Key, Val)`) and sends it on to the customer (`msg(C, A, lookupReply(Key, Val))`).

Finally, the rule labeled `tell` specifies how an agent handles a `tellReq` message. An agent can handle such a request in any state, as indicated by the status variable `st : Any`. The agent simply updates its `data` attribute and acknowledges the request, including the request parameters in the `tellReply` message.

Note that in the case of agents with just these three attributes, using the `AttributeSet` variable `Atts` or the status variable `st : Any` are equivalent ways of saying that the rule matches any status attribute. 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.

Recall the assumption that a key is set (via any `tellReq` message) at most once. This means that if an agent's data dictionary is initially empty, then it will always contain at most one value for any key. Furthermore, if the dictionaries of two agents each contain a value for a key (other than `'nothing`) then they contain the same value.

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.

```
mod AGENT-TEST is
  ex DATA-AGENTS .
  ops sam joe me : -> Oid [ctor] .
```

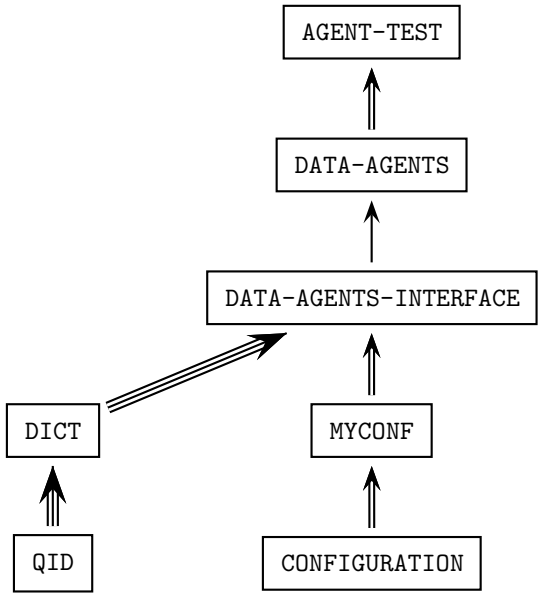


Figure 8.3: Importation graph of data-agents modules.

```
  op iconf : -> Configuration .
  eq iconf =
       < sam : DataAgent | data : none, pal : joe, st : idle >
       msg(sam, me, tellReq('a, 'bc))
       msg(sam, me, tellReq('d, 'ef))
       < joe : DataAgent | data : none, pal : sam, st : idle >
       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, resulting in a configuration in which the agents have their initial data, and there are replies for each received `tellReq`.

```
Maude> rew iconf .
rewrite in AGENT-TEST : iconf .
rewrites: 9 in 0ms cpu (1ms real) (~ rewrites/second)
result Configuration:
  < sam : DataAgent | st : idle, data : (r('d, 'ef) r('a, 'bc)), pal : joe >
  < joe : DataAgent | st : idle, data : (r('g, 'hi) r('j, 'kl)), pal : sam >
  msg(me, sam, tellReply('d, 'ef))
  msg(me, sam, tellReply('a, 'bc))
  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 `'nothing`.

```
Maude> rew iconf msg(sam, me, lookupReq('a)) .
rewrite in AGENT-TEST : iconf msg(sam, me, lookupReq('a)) .
rewrites: 17 in 0ms cpu (1ms real) (~ rewrites/second)
result Configuration:
  < sam : DataAgent | st : idle, data : (r('d, 'ef) r('a, 'bc)), pal : joe >
  < joe : DataAgent | st : idle, data : (r('g, 'hi) r('j, 'kl)), pal : sam >
  msg(me, sam, tellReply('d, 'ef))
  msg(me, sam, tellReply('a, 'bc))
  msg(me, sam, lookupReply('a, 'nothing))
  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 `'nothing`. Note the use of the `such that` condition to filter search solutions (see Section 15.4).

```
Maude> search iconf msg(sam, me, lookupReq('a))
         =>! C:Configuration msg(me, sam, lookupReply('a, Q:Qid))
         such that Q:Qid =/= 'nothing = true .
search iconf msg(sam, me, lookupReq('a))
  =>! C:Configuration msg(me, sam, lookupReply('a, Q:Qid))
  such that Q:Qid =/= 'nothing = true .

Solution 1 (state 69)
states: 78  rewrites: 418 in 30ms cpu (31ms real) (13933 rewrites/second)
C:Configuration -->
  < sam : DataAgent | st : idle, data : (r('d, 'ef) r('a, 'bc)), pal : joe >
  < joe : DataAgent | st : idle, data : (r('g, 'hi) r('j, 'kl)), pal : sam >
  msg(me, sam, tellReply('d, 'ef))
  msg(me, sam, tellReply('a, 'bc))
  msg(me, joe, tellReply('g, 'hi))
  msg(me, joe, tellReply('j, 'kl))
Q:Qid --> 'bc

No more solutions.
states: 80  rewrites: 436 in 40ms cpu (37ms real) (10900 rewrites/second)
```

Indeed, there is just one such reply.

8.4 External objects

This section explains Maude's support for rewriting with external objects and an implementation of sockets 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`. Rewriting with external objects is started by the command `erewrite` (abbreviated `erew`) which is like `frewrite` (see Sections 5.4 and 8.2) except it allows messages to be exchanged with external objects that do not reside in the configuration. Currently the command `erewrite` has some serious limitations:

1. Limit/gas parameters and continuation do not work.
2. Rewrites that involve messages entering or leaving the configuration do not show up in tracing, profiling or rewrite counts.
3. Bad interactions with break points and debugger.
4. Potential race condition with `^C`.

Note that `erewrite` may not terminate just because there are no more rewrites possible; 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 a command `erewrite` do not survive to the next `erewrite`. If a message to an external object is malformed or inappropriate or the external object is not ready for it, it just stays in the configuration for future acceptance or for debugging purposes.

The first example of external objects is *sockets*, which are accessed using the messages declared in the following 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 8.4.1 which solves this problem.

The following modules implement an updated version of the five rule HTTP/1.0 client from the paper "Towards Maude 2.0" [14] that is now executable. The first module defines some auxiliary operations on strings.

```
fmod STRING-OPS is
  pr 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 to a HTTP server.

```
mod HTTP/1.0-CLIENT is
  pr STRING-OPS .
  inc 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 special 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:

```
  erew start("www.google.com") .
```

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 myClass : -> Cid .
  ops myObj : -> Oid .

  var O O1 O2 O3 : Oid .
  var A : AttributeSet .
  var N : Nat .
  var 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 : myClass | A > createdSocket(O, O2, O3)
    => < O : myClass | A > acceptClient(O3, O) .

  rl [acceptedClient] :
    < O : myClass | A > acceptedClient(O, O2, S, O3)
    => < O : myClass | A > receive(O3, O) acceptClient(O2, O) .
```

```
    rl [received] :
      < O : myClass | A > received(O, O2, S)
      => < O : myClass | A > send(O2, O, string(rat(S, 10)!, 10)) .

    rl [sent] :
      < O : myClass | A > sent(O, O2)
      => < O : myClass | A > closeSocket(O2, O) .

    rl [closedSocket] :
      < O : myClass | A > closedSocket(O, O2, S)
      => < O : myClass | A > .
endm
```

The Maude command that initializes the server is as follows, where the configuration includes the portal <>.

```
erew <> < myObj : myClass | none >
     createServerTcpSocket(socketManager, myObj, 8811, 5) .
```

The second module describes the behavior of the clients.

```
mod FACTORIAL-CLIENT is
  inc SOCKET .
  op myClass : -> Cid .
  op myObj : -> Oid .

  var O O1 O2 O3 : Oid .
  var A : AttributeSet .
  var N : Nat .
```

Using the following rules, the client connects to the server (clients must be created after the server), sends a message representing a number[3], and then waits for the response. When the response arrives, there are no blocking messages and rewriting ends.

```
    rl [createdSocket] :
      < O : myClass | A > createdSocket(O, O2, O3)
      => < O : myClass | A > send(O3, O, "6") .

    rl [sent] :
      < O : myClass | A > sent(O, O2)
      => < O : myClass | A > receive(O2, O) .
endm
```

The initial configuration for the client will be as follows, again with portal <>.

```
erew <> < myObj : myClass | none >
     createClientTcpSocket(socketManager, myObj, "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.

[3] In this quite simple example, it is always "6".

8.4.1 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 the confusion. Thus, to create a client buffered socket, you send `socketManager` a message

```
CreateClientTcpSocket(socketManager, ME, ADDRESS, PORT)
```

instead of a message

```
createClientTcpSocket(socketManager, ME, ADDRESS, PORT).
```

All the messages have exactly the same declarations, being the only difference 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, being the boundary control 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 say 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`.

```
sort BufferedSocket .
subsort BufferedSocket < Cid .
op BufferedSocket : -> BufferedSocket [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 to 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 `BufferedSocket` object understands capitalized versions of all the messages a socket object understands. For most of them, it just converts them into the corresponding uncapitalized message. There are messages `CreateServerTcpSocket`, `AcceptClient`, `CloseSocket` 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) .
```

A `BufferedSocket` object can convert too an uncapitalized message into the capitalized one. The rule `socketError` shows it:

```
rl [socketError] :
  socketError(O, SOCKET-MANAGER, REASON)
  => SocketError(O, SOCKET-MANAGER, REASON) .
```

Note that in these cases the buffered-socket versions of the messages are just translated into the corresponding socket messages.

`BufferedSocket` objects are created and destroyed when the corresponding sockets. 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) : V@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[4] to indicate the end of the message.

```
rl [send] :
  < b(SOCKET) : V@BufferedSocket | bState : active, Atts >
  Send(b(SOCKET), O, DATA)
  => < b(SOCKET) : V@BufferedSocket | bState : active, Atts >
     send(SOCKET, O, DATA + "#") .

rl [sent] :
  < b(SOCKET) : V@BufferedSocket | bState : active, Atts >
  sent(O, SOCKET)
  => < b(SOCKET) : V@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, 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) : V@BufferedSocket | bState : idle, Atts >
  => < b(SOCKET) : V@BufferedSocket | bState : active, Atts >
     receive(SOCKET, b(SOCKET)) .

rl [received] :
  < b(SOCKET) : V@BufferedSocket | bState : active, read : S, Atts >
  received(b(SOCKET), O, DATA)
  => < b(SOCKET) : V@BufferedSocket | bState : active, read : (S + DATA), Atts >
     receive(SOCKET, b(SOCKET)) .

crl [Received] :
  < b(SOCKET) : V@BufferedSocket | bState : active, read : S, Atts >
  Receive(b(SOCKET), O)
  => < b(SOCKET) : V@BufferedSocket | bState : active, read : S', Atts >
     Received(O, b(SOCKET), DATA)
  if N := find(S, "#", O)
```

[4] We use the character '#' as mark, so that the user data sent through the sockets should not contain such a character.

```
    /\ DATA := substr(S, 0, N)
    /\ S' := substr(S, N + 1, length(S)) .
```

The BUFFERED-SOCKET module is part of Mobile Maude, a mobile agent language based on Maude, whose distributed implementation is currently under way.

Chapter 9

Model Checking

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, 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 [11]; we focus on linear temporal logic [41, 11], because of its intuitive appeal, widespread use, and well-developed proof methods and decision procedures.

9.1 LTL formulae and the `LTL` module

Given a set AP of *atomic propositions*, we define the formulae 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 formulae $\neg\varphi$, 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
  sorts Prop Formula .
  subsort Prop < 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 the subsort `Prop` of `Formula`, corresponding to the set AP of atomic propositions. For the moment this is left unspecified. Section 9.2 explains how such atomic propositions are defined for a given system module `M`. 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.

9.2 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 the temporal logic LTL 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 $s; \pi(n) = \pi(s(n))$.
- 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 $\varphi \vee \psi \in \mathrm{LTL}(AP)$,
$$\mathcal{A}, \pi \models_{LTL} \varphi \vee \psi \quad \Leftrightarrow$$
$$\mathcal{A}, \pi \models_{LTL} \varphi \quad \text{or} \quad \mathcal{A}, \pi \models_{LTL} \psi.$$

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` (which is contained in the file `model-checker.maude`) is very simple, and has the following specification:

```
fmod SATISFACTION is
  protecting LTL .
  sort State .
  op _|=_ : State Formula ~> Bool .
endfm
```

where the sort `State` is 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 Formula ~> Bool .
```

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 .
  op __ : Conf Conf -> Conf [assoc comm id: none] .
  ops a b : -> Name .
  ops wait critical : -> Mode .
  op [_,_] : Name Mode -> Proc .
  ops * $ : -> Token .
  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 .
  eq [N, critical] C |= crit(N) = true .
  eq [N, wait] C |= wait(N) = true .
endm
```

Note the two 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.

Note that *only the cases when a predicate holds* need to be specified: given a state t and a, possibly parametric, state predicate $p(u_1, \ldots, u_n)$, when the ground expression $t \models p(u_1, \ldots, u_n)$ cannot be simplified to *true*, then the predicate does *not* hold. This means that to specify the semantics of the state predicates it is enough to give (possibly conditional) equations of the general form

$$t \models p(v_1, \ldots, v_n) = true \;\; \textit{if} \;\; C.$$

There is in principle no need to specify when a state predicate is *false*. However, if so desired one can specify both the true and false cases. This can always be easily done either by using the `[owise]` attribute, or by giving (possibly conditional) equations of the more general form

$$t \models p(v_1, \ldots, v_n) = bexp \;\; \textit{if} \;\; C,$$

where *bexp* is an arbitrary Boolean expression.

We are now ready to associate to 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

$$L_\Pi([t]) = \{\theta(p) \in AP_\Pi \mid (E \cup D) \vdash \ (\forall\, \emptyset)\ t \models \theta(p) = true\}.$$

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 (perhaps modulo some axioms A contained in E) and R is (ground) coherent relative to E, u can always be chosen to be the canonical form of t under the equations E.

9.3 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 9.2.

Then, as explained in Section 9.2, 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, 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 [11]. 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 to) $(\mathcal{K}(\mathcal{R}, k)_\Pi, [t])$. 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 the synchronous product. For a detailed explanation of this general method of on-the-fly model LTL checking, see [11]; and for a discussion of the specific techniques used in the Maude LTL model-checker implementation, see [36].

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 $B_{\neg\varphi}$.

[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.

```
fmod LTL-SIMPLIFIER is
  including LTL .

  *** The simplifier is based on:
  ***   Kousha Etessami and Gerard J. Holzman,
  ***   "Optimizing Buchi Automata", p153-167, 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 .

  *** 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 .
  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.

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
  including MUTEX-PREDS .
  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 9.1, 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))) .
rewrites: 18 in 10ms cpu (10ms real) (1800 rewrites/second)
result Bool: true
```

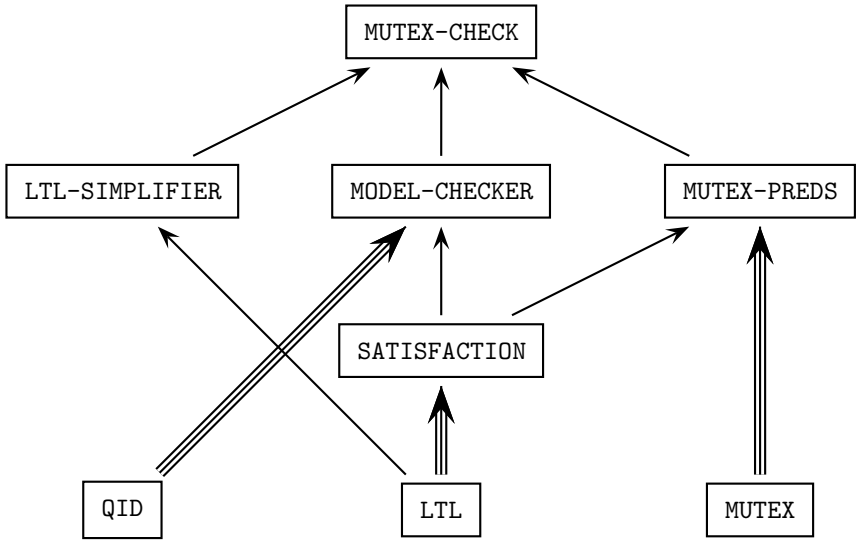


Figure 9.1: Importation graph of model-checking modules.

```
Maude> red modelCheck(initial2, [] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []~ (crit(a) /\ crit(b))) .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
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)) .
rewrites: 76 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> red modelCheck(initial1, ([] <> wait(b)) -> ([] <> crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial1, []<> wait(b) -> []<> crit(b)) .
rewrites: 76 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> red modelCheck(initial2, ([] <> wait(a)) -> ([] <> crit(a))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []<> wait(a) -> []<> crit(a)) .
rewrites: 68 in 10ms cpu (10ms real) (6800 rewrites/second)
result Bool: true

Maude> red modelCheck(initial2, ([] <> wait(b)) -> ([] <> crit(b))) .
reduce in MUTEX-CHECK : modelCheck(initial2, []<> wait(b) -> []<> crit(b)) .
rewrites: 68 in 0ms cpu (0ms real) (~ rewrites/second)
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)) .
rewrites: 14 in 0ms cpu (1ms real) (~ rewrites/second)
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 [11]. 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 BAD-EXAMPLE is
  including MODEL-CHECKER .
  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 `a` $\rightarrow$ `f(a)` $\rightarrow$ `f(f(a))` $\rightarrow$ `f(f(f(a)))` $\rightarrow \cdots$, 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 BAD-EXAMPLE : 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.

9.4 The LTL satisfiability and tautology checker

A formula $\varphi \in \mathrm{LTL}(AP)$ is *satisfiable* if there is a Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$, 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
  including 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
```

One can define the desired atomic predicates in a module extending `SAT-SOLVER`, such as, for example,

```
fmod TEST is
  including SAT-SOLVER .
  ops a b c d e p q r : -> Prop .
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 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 TEST : satSolve(O O (~ c /\ [](c \/ O c)) /\ (a /\ O b)) .
rewrites: 2 in 10ms cpu (10ms real) (200 rewrites/second)
result SatSolveResult: model(a ; b, (~ c) ; c)
```

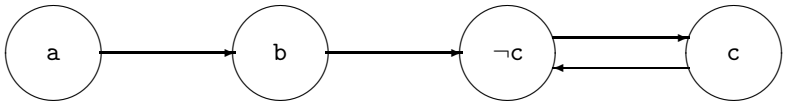


Figure 9.2: Graphical representation of a Kripke structure.

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 9.2.

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 TEST : tautCheck((a => O a) <-> a => []a) .
rewrites: 49 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> red tautCheck((a -> (O a)) <-> (a -> ([] a))) .
reduce in TEST : tautCheck((a -> O a) <-> a -> []a) .
rewrites: 33 in 10ms cpu (10ms real) (3300 rewrites/second)
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 [36].

Chapter 10

Reflection, Metalevel Computation, and 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 [18, 12, 19, 20]. 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.

10.1 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:

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

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 a functional module `META-LEVEL`. This module includes the modules `META-MODULE` 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-MODULE`, Maude modules are metarepresented as terms in a data type `Module` of modules; and
- in the module `META-LEVEL`,
  - the process of reducing a term to canonical form using Maude's `reduce` command is metarepresented by a built-in function `metaReduce`;
  - 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 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 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`; 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.

We call the functions `metaReduce`, `metaApply`, `metaXapply`, `metaRewrite`, `metaFrewrite`, `metaMatch`, and `metaXmatch`, *descent functions*, since they allow us to descend levels in the reflective tower. The paper [15] 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 10.1 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 7.11.1), and the module `QID-SET` provides sets of quoted identifiers (see Section 7.11.2). Notice that `QID-SET` is imported (in protecting mode) with renaming `(op empty to none, op _,_ to _;_ [prec 43])` abbreviated to β in the figure.

10.2 The `META-TERM` module

10.2.1 Metarepresenting sorts and kinds

In the module `META-TERM` sorts and kinds are metarepresented as specific subsorts of the sort `Qid` of quoted identifiers.

A term of sort `Sort` is any quoted identifier not containing the following characters: '`:`', '`.`', '`[`', and '`]`'. The characters '`{`', '`}`', and '`,`' in structured sorts (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

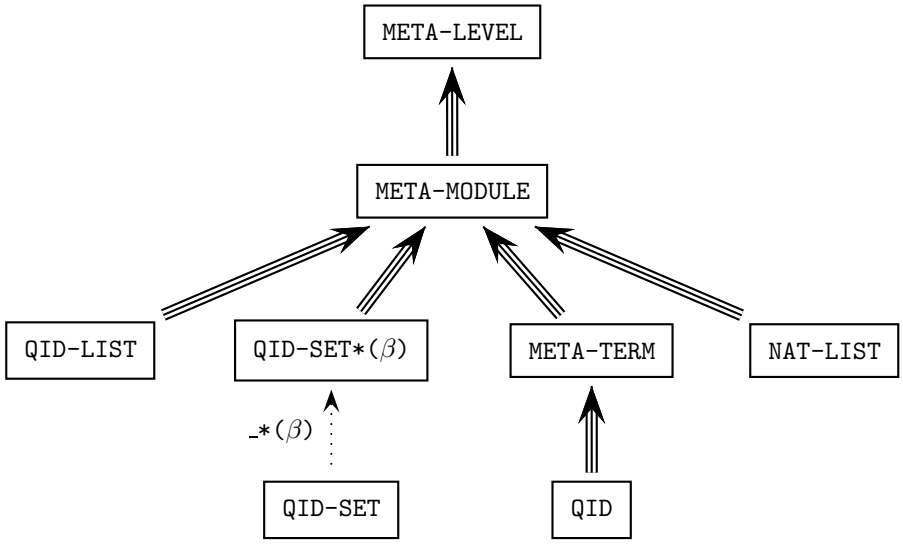


Figure 10.1: Importation graph of meta-level modules.

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. Commas 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 to 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.

10.2.2 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`]
```

Then a term is constructed in the usual way, by applying an operator symbol to a list of terms.

```
sort TermList .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList [ctor assoc gather (e E) prec 120] .
op _[_] : Qid TermList -> Term [ctor] .
```

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 `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)`.

10.3 The `META-MODULE` module: Metarepresenting modules

In the module `META-MODULE`, which imports `META-TERM`, functional and system modules, as well as functional and system 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 10.2.2;

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 functional and system modules and functional and system theories (just modules in general) 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 FTheory STheory Module .
subsorts FModule < SModule < Module .
subsorts FTheory < STheory < Module .
sort Header .
subsort Qid < Header .
op _{_}  : Qid ParameterDeclList -> Header .
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 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 (& & & & & & &)] .
```

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 .
```

Without going into all the syntactic details, we show only the operators used to metarepresent sets of sorts and kinds, conditions, equations, and rules. The complete syntax used for metarepresenting modules can be found in the module `META-MODULE` in the file `prelude.maude`.

```
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 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] .

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] .
```

For example, we show here the metarepresentations of the VENDING-MACHINE-SIGNATURE and VENDING-MACHINE modules introduced in Section 5.1.

```
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 s d d d)] .
op label_to_ : Qid 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 'FOO{'X :: 'TRIV} is
  including 'MAP{'String, 'X} .
  sorts 'Foo .
  none
  none
  none
  none
endfm
```

Views are not reflected; there are no metaviews and no way to construct new views or inspect existing views at the metalevel. Therefore, the views used in the module expressions occurring in metamodules must be in the database.

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.

10.4 The `META-LEVEL` module: Metalevel operations

The `META-LEVEL` module, which imports `META-MODULE`, 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 metarepresentations of modules or are not metarepresentations of terms in the module provided as another argument.

10.4.1 Moving between reflection levels: `upModule`, `upTerm`, `downTerm`, and others

For a module $\mathcal{R}$ *already loaded*, the operations `upModule`, `upSorts`, `upSubsortDecl`, `upOpDecls`, `upMbs`, `upEqs`, and `upRls` 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, of its subsort declarations, of its operator declarations, of its membership axioms, of its equations, and ofits rules. If the second argument of these functions is `true`, 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 ( ... )] .
```

These are 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 a `protecting` importation 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 module `VENDING-MACHINE` from Chapter 5, have already been loaded into Maude.

```
Maude> reduce in META-LEVEL : upModule('VENDING-MACHINE-SIGNATURE, false) .
rewrites: 1 in 0ms cpu (1ms real) (~ rewrites/second)
result FModule:
  fmod 'VENDING-MACHINE-SIGNATURE is
  protecting '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) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
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) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result EquationSet: (none).EquationSet

Maude> reduce in META-LEVEL : upRls('VENDING-MACHINE, true) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
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 to use an already loaded module at the metalevel. This operator is defined in the module `META-MODULE` as follows:

```
op [_] : Qid -> Module .
eq [Q:Qid] = (th Q:Qid is including Q:Qid .
              sorts none . none none none none none endth) .
```

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 ( ... )] .
```

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
  including 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))) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Foo: f(a, f(b, d))

Maude> reduce in UP-DOWN-TEST : downTerm(upTerm(f(a, f(b, c))), error) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
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.
rewrites: 1 in 1ms cpu (10ms real) (1000 rewrites/second)
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]`.

10.4.2 Simplifying and rewriting: `metaReduce`, `metaRewrite`, and `metaFrewrite`

`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(`$\overline{\mathcal{R}}$`,`$\overline{t}$`)` 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 (Section 15.2).

As mentioned above in general, when either the first argument of `metaReduce` is a term of sort `Module` but not a metarepresentation $\overline{\mathcal{R}}$ of an object module $\mathcal{R}$, or the second argument is not the metarepresentation $\overline{t}$ 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 a sort, but only 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]]) .
```

```
rewrites: 447 in 0ms cpu (0ms real) (~ rewrites/second)
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]],
        'IntList}
```

We can also insert a new element to an empty map of the type declared in the module `FOO` at the end of Section 10.3 as follows:

```
Maude> red in META-LEVEL :
         metaReduce(
           fmod 'FOO{'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`.

`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 (Section 15.2). 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) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
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) .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
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`$(\overline{\mathcal{R}}, \overline{t}, b, n)$ 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) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'__['$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin], 'Marking}

Maude> reduce in META-LEVEL :
         metaFrewrite(upModule('VENDING-MACHINE, false),
           '__['$.Coin, '__['$.Coin, '__['q.Coin, 'q.Coin]]], 12, 1) .
rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
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}
```

10.4.3 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`$(\overline{\mathcal{R}},\ \overline{t},\ \overline{l},\ \sigma,\ n)$ 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 for error recovery for example.

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) .
rewrites: 3 in 0ms cpu (370ms real) (~ rewrites/second)
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) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultTriple:
  {'__['$.Coin, '$.Coin], 'Marking, 'M:Marking <- '$.Coin}
```

However, using `metaApply`, we cannot force the rewriting of the term `(q $)` with the rule `buy-c`, since its lefthand side, `$`, 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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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`$(\overline{\mathcal{R}}, \overline{t}, \overline{l}, \sigma, n, b, m)$ is evaluated as `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.

As in the `metaApply` case, the last `Nat` argument m in `metaXapply`$(\overline{\mathcal{R}}, \overline{t}, \overline{l}, \sigma, n, b, m)$ 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 `CTermList` represents 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 `CTermList` and `TermList` needed for the `assoc` attribute to make sense.

```
sorts Context CTermList GTermList  .
subsort Context < CTermList .
subsorts TermList CTermList < GTermList .
op [] : -> Context [ctor] .
op _,_ : TermList CTermList -> CTermList [ctor assoc gather (e E) prec 120] .
op _,_ : CTermList TermList -> CTermList [ctor assoc gather (e E) prec 120] .
op _[_] : Qid CTermList -> 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) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Result4Tuple?: (failure).Result4Tuple?
```

10.4.4 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`. These two operations provide respectively the metalevel counterparts of the object level commands `match` and `xmatch` (Section 15.3).

```
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 .
```

As examples, we can 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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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 in META-LEVEL :
         metaMatch(upModule('VENDING-MACHINE, false),
           '__['M:Marking, '$.Coin],
           '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
           nil, 1) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Substitution?: (noMatch).Substitution?
```

- anywhere, asking for the first solution,

```
Maude> reduce in META-LEVEL :
         metaXmatch(upModule('VENDING-MACHINE, false),
           '__['M:Marking, '$.Coin],
           '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
           nil, 0, unbounded, 0) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result MatchPair:
  {'M:Marking <- '__['q.Coin, 'a.Item, 'c.Item], []}
```

- anywhere, asking for the second solution,

```
Maude> reduce in META-LEVEL :
         metaXmatch(upModule('VENDING-MACHINE, false),
           '__['M:Marking,'$.Coin],
           '__['$.Coin,'q.Coin,'a.Item,'c.Item],
```

```
            nil, 0, unbounded, 1) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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 in META-LEVEL :
         metaMatch(upModule('VENDING-MACHINE, false),
           '__['M:Marking,'$.Coin],
           '__['$.Coin,'q.Coin,'a.Item,'c.Item],
           'M:Marking = 'a.Item, 0) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Substitution?: (noMatch).Substitution?
```

- anywhere, asking for the first solution satisfying a given condition,

```
Maude> reduce in META-LEVEL :
         metaXmatch(upModule('VENDING-MACHINE, false),
           '__['M:Marking,'$.Coin],
           '__['$.Coin,'q.Coin,'a.Item,'c.Item],
           'M:Marking = 'a.Item, 0, unbounded, 0) .
rewrites: 2 in 10ms cpu (10ms real) (200 rewrites/second)
result MatchPair:
  {'M:Marking <- 'a.Item,'__['__['q.Coin,'c.Item],[]]}
```

10.4.5 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 (Section 15.4). 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 1 (corresponding to `=>1` in the `search` command).

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 a 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) .
rewrites: 1356 in 10ms cpu (26ms real) (135600 rewrites/second)
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) .
rewrites: 505 in 0ms cpu (5ms real) (~ rewrites/second)
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),
           '__['$.Coin, 'q.Coin, 'q.Coin,'q.Coin],
           '__['c.Item, 'a.Item, 'c.Item, 'M:Marking],
           nil, '+, unbounded, 0) .
rewrites: 1356 in 10ms cpu (39ms real) (135600 rewrites/second)
result Trace:
  {'__['$.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),
           '__['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
           '__['c.Item, 'a.Item, 'c.Item, 'M:Marking],
           nil, '+, unbounded, 1) .
rewrites: 505 in 0ms cpu (14ms real) (~ rewrites/second)
result Trace:
  {'__['$.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.

10.4.6 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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'__['$.Coin,'__['q.Coin,'__['q.Coin,'q.Coin]]],'Marking}

Maude> reduce in META-LEVEL :
   metaParse(upModule('VENDING-MACHINE, false), '$ 'q 'd 'q, 'Marking) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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. Also, 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 15.6, 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),
           '__['$.Coin, '__['q.Coin, '__['q.Coin, 'q.Coin]]], none) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NeQidList: '__ '`( '$ '`, '__ '`( 'q '`, '__ '`( 'q '`, 'q '`) '`) '`)

Maude> reduce in META-LEVEL :
         metaPrettyPrint(upModule('VENDING-MACHINE, false),
           '__['$.Coin, '__['q.Coin, '__['q.Coin, 'q.Coin]]], mixfix) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NeTypeList: '$ 'q 'q 'q

Maude> reduce in META-LEVEL :
         metaPrettyPrint(upModule('VENDING-MACHINE, false),
           '__['$.Coin, '__['q.Coin, '__['q.Coin, 'q.Coin]]], mixfix format) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NeTypeList: '\r '\! '$ '\o '\r '\! 'q '\o '\r '\! 'q '\o '\r '\! 'q '\o
```

Notice how `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),
           '__['$.Coin, '__['q.Coin, '__['q.Coin, 'q.Coin]]]) .
rewrites: 2 in 0ms cpu (22ms real) (~ rewrites/second)
result NeTypeList: '\r '\! '$ '\o '\r '\! 'q '\o '\r '\! 'q '\o '\r '\! 'q '\o
```

10.4.7 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 `gblSorts` 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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> reduce in META-LEVEL :
  sortLeq(upModule('NUMBERS, false), 'Zero, 'NzNat) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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`]) .
rewrites: 2 in 0ms cpu (1ms real) (~ rewrites/second)
result Bool: true

Maude> reduce in META-LEVEL :
   sortLeq(upModule('NUMBERS, false), '`[Zero`], '`[Bool`]) .
rewrites: 2 in 0ms cpu (7ms real) (~ rewrites/second)
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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false

Maude> reduce in META-LEVEL :
   sortLeq(upModule('NUMBERS, false), 'Zero, '`[NatSet`]) .
rewrites: 2 in 0ms cpu (1ms real) (~ rewrites/second)
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, 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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> reduce in META-LEVEL :
   sameKind(upModule('NUMBERS, false), 'Zero, 'Nat3) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false

Maude> reduce in META-LEVEL :
   sameKind(upModule('NUMBERS, false), ''[Zero`], ''[NzNat`]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> reduce in META-LEVEL :
   sameKind(upModule('NUMBERS, false), ''[Zero`], 'NzNat) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Sort: 'Zero

Maude> reduce in META-LEVEL :
   completeName(upModule('NUMBERS, false), ''[Zero`]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Kind: ''[NatSeq‘,NatSet‘]

Maude> reduce in META-LEVEL : getKind(upModule('NUMBERS, false), ''[Zero‘]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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)) .
rewrites: 2 in 0ms cpu (46ms real) (~ rewrites/second)
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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NeSortSet: 'NzNat ; 'Zero

Maude> reduce in META-LEVEL : lesserSorts(upModule('NUMBERS, false), 'Zero) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL : lesserSorts(upModule('NUMBERS, false), 'NatSeq) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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`]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NeSortSet: 'Nat ; 'NatSeq ; 'NatSet ; 'NzNat ; 'Zero

Maude> reduce in META-LEVEL : lesserSorts(upModule('NUMBERS, false), '`[Bool`]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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.

```
op leastSort : Module Term ~> Type [special ( ... )] .
```

For example,

```
Maude> reduce in META-LEVEL :
   leastSort(upModule('NUMBERS, false), 'p['s_['zero.Zero]]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Sort: 'Nat
```

`glbSorts`

The operation `gblSorts` 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 `gblSorts` 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) .
  rewrites: 2 in 0ms cpu (31ms real) (~ rewrites/second)
  result Sort: 'Zero

  Maude> reduce in META-LEVEL :
     glbSorts(upModule('NUMBERS, false), 'NatSet, 'NatSeq) .
  rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
  result Sort: 'Nat

  Maude> reduce in META-LEVEL :
     glbSorts(upModule('NUMBERS, false), 'NzNat, 'NzNat) .
  ```

```
rewrites: 2 in 0ms cpu (1ms real) (~ rewrites/second)
result Sort: 'NzNat

Maude> reduce in META-LEVEL :
   glbSorts(upModule('NUMBERS, false), 'Zero, 'NzNat) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
   glbSorts(upModule('NUMBERS, false), 'NzNat, 'Bool) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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']) .
rewrites: 2 in 0ms cpu (46ms real) (~ rewrites/second)
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
   glbSorts(upModule('NUMBERS, false), ''[Nat'], ''[NatSeq']) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
   glbSorts(upModule('NUMBERS, false), ''[NatSeq'], 'Zero) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Sort: 'Zero

Maude> reduce in META-LEVEL :
   glbSorts(upModule('NUMBERS, false), 'NzNat, ''[NatSet']) .
rewrites: 2 in 0ms cpu (3ms real) (~ rewrites/second)
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']) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NeSortSet: 'NatSeq ; 'NatSet

Maude> reduce in META-LEVEL :
   minimalSorts(upModule('NUMBERS, false), ''[Zero']) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result TypeListSet: 'Nat 'NzNat ; 'NzNat 'Nat

Maude> reduce in META-LEVEL :
```

```
   maximalAritySet(upModule('NUMBERS, false), '_+_, 'Nat 'Nat, 'NzNat) .
rewrites: 2 in 10ms cpu (5ms real) (200 rewrites/second)
result TypeListSet: 'Nat 'NzNat ; 'NzNat 'Nat

Maude> reduce in META-LEVEL :
   maximalAritySet(upModule('NUMBERS, false), '_+_, 'Nat 'Nat, 'Nat) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NeTypeList: 'Nat 'Nat

Maude> reduce in META-LEVEL :
   maximalAritySet(upModule('NUMBERS, false), '_+_, 'Nat3 'Nat3, 'Nat3) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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 ellided the lengthy metarepresentation of the `NUMBERS` module.

```
Maude> reduce in META-LEVEL :
   maximalAritySet(upModule('NUMBERS, false), '_+_, 'Nat3 'Nat3, 'NzNat) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result [GTermList,ParameterList,QidList,TypeListSet,Type?,ModuleExpression,Header]:
  maximalAritySet(fmod 'NUMBERS is ... endfm, '_+_, 'Nat3 'Nat3, 'NzNat)
```

10.4.8 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)) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> reduce in META-LEVEL :
   wellFormed(upModule('NUMBERS, false), 'p['zero.Zero]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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.
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false

Maude> reduce in META-LEVEL :
   wellFormed(upModule('NUMBERS, false), 'N:Zero <- 'zero.Zero) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> reduce in META-LEVEL :
   wellFormed(upModule('NUMBERS, false), 'N:Nat <- 'p['zero.Zero]) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
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.
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false
```

10.5 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*. 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 the possibilities by implementing the following strategies for controlling the execution of the rules in the module `VENDING-MACHINE` 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 is an extension of the `META-LEVEL` module.

```
fmod BUYING-STRATS is
  including META-LEVEL .
  protecting NAT .
```

The function `insertCoin` below defines the strategy (1): it expects the metarepresentation of a coin and the metarepresentation of the marking of a vending machine, and it applies the corresponding rule once. The rules `add-q` and `add-$` are applied using the built-in function `metaXapply`. Note the use of the statement attribute `owise` (see Section 4.5.4) to define the function `insertCoin` for unexpected cases.

```
op insertCoin : Qid Term -> Term .
var T : Term .
var Q : Qid .

eq insertCoin('q.Coin, T)
   = getTerm(metaXapply(upModule('VENDING-MACHINE, false), T,
                        'add-q, none, 0, unbounded, 0)) .
eq insertCoin('$.Coin, T)
   = getTerm(metaXapply(upModule('VENDING-MACHINE, false), T,
                        'add-$, 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 metarepresentation of the current marking of the vending machine is returned. The rules `buy-c` and `change` are applied using the built-in function `metaXapply`. A rule cannot be applied anymore when the result of `metaXapply`-ing the rule is not a term of sort `Result4Tuple`. Note the use of matching equations in the condition to simplify the presentation of the righthand side of the equation (see Section 4.3).

```
op onlyCakes : Term -> Term .
vars BuyCake? Change? : Result4Tuple? .

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 before, the rules are applied using the built-in function `metaXapply`. Note the use of the symmetric difference operator `sd` (see Section 7.2) to decrement `N`.

```
op onlyNitems : Term Qid Nat -> Term .
var N : Nat .
```

```
  var BuyItem? : Result4Tuple? .

  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) .
```

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 cakesAndApples(T)
    = if buyItem?(T, 'buy-c)
      then (if buyItem?(onlyNitems(T, 'buy-c, 1), 'buy-a)
            then cakesAndApples(onlyNitems(onlyNitems(T, 'buy-c, 1), 'buy-a, 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]) .
rewrites: 32 in 0ms cpu (0ms real) (~ rewrites/second)
result GroundTerm: '__['q.Coin,'c.Item,'c.Item,'c.Item]

Maude> reduce in BUYING-STRATS :
         onlyNitems('__['$.Coin, '$.Coin, '$.Coin, 'q.Coin], 'buy-a, 3) .
rewrites: 64 in 0ms cpu (0ms real) (~ rewrites/second)
result GroundTerm: '__['q.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'a.Item, 'a.Item, 'a.Item]

Maude> reduce in BUYING-STRATS :
         cakesAndApples('__['$.Coin, '$.Coin, '$.Coin, 'q.Coin]) .
rewrites: 148 in 0ms cpu (0ms real) (~ rewrites/second)
result GroundTerm: '__['$.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. See [12] for a general methodology for defining internal strategy languages using reflection, and [13, 15] for other examples of rewriting strategies defined in Maude.

Chapter 11

User Interfaces and Metalanguage Applications

11.1 The LOOP-MODE module

Using object-oriented concepts, we can specify in Maude a general input/output facility provided by the LOOP-MODE module shown below, which extends the module QID-LIST (see Section 7.9), 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 may not be maintained in future versions. As mentioned in the introduction, we plan to endow Maude with support for communication with external objects; this will make possible more general and flexible solutions for dealing with input/output.

Since in the current release 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 is written in the Maude prompt enclosed in parentheses 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. 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 printed on the terminal after applying the inverse process of "unquoting" each of the identifiers in the list. 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 constructor. 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 structure of its state and the rewrite rules for changing the state and interacting with the loop.

11.2 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 a data type $Sign_{\mathcal{P}}$ for commands and other constructs.

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 the interaction with the vending machine. The signature of this module extends the signature of `VENDING-MACHINE-SIGNATURE` with operators to represent the valid actions: namely, `$` and `q` for inserting a dollar or a quarter in the machine; `showBasket` and `showCredit` for showing the items already bought or the remaining credit; `buy1Apple` and `buy1Cake` for buying an apple or a cake; and `buy_:_` for buying a number of pieces of the same item.

```
fmod VENDING-MACHINE-GRAMMAR is
  protecting VENDING-MACHINE-SIGNATURE .
  protecting NAT .
  sort Action .
  op $ : -> Action .
  op q : -> Action .
  op showBasket : -> Action .
  op showCredit : -> Action .
  op buy1Cake : -> Action .
  op buy1Apple : -> Action .
  op buy_:_ : Item Nat -> Action .
endfm
```

Next, we define in an extension of the `LOOP-MODE` module the terms of sort `State` representing the state of the loop for this application. In the module `VENDING-MACHINE-INTERFACE` below we define this state as a triple: its first element is the next action requested by the client (inserting a coin, showing information about the remaining credit or the items already bought, or buying one or more items); its second element is the current state of the machine (the *marking* of the vending machine, that is, the remaining credit plus the items already bought); and its third element represents the response of the machine to the last action requested by the client. The response of the vending machine will have the form of a message, which can be represented as a list of quoted identifiers. The constant `init` denotes the initial state of the whole system, including the empty input and output streams and the "empty" initial state of the vending machine.

```
mod VENDING-MACHINE-INTERFACE is
```

```
including LOOP-MODE .
including VENDING-MACHINE-GRAMMAR .
protecting BUYING-STRATS .
protecting CONVERSION .

op <_;_;_> : Action Marking QidList -> State .
op init : -> System .
op idle : -> Action .
eq init = [nil, < idle ; null ; nil >, nil] .

var  A : Action .
var  I : Item .
var  C : Coin .
var  M : Marking .
vars QIL QIL' QIL'' : QidList .
var  N : Nat .
```

Now we define in this module, using rewrite rules, the interaction of the state of the vending machine with the loop—and, consequently, with the client—and the changes produced in the state of the vending machine by the actions requested by the client. As explained before, the client will request an action by enclosing the text in parentheses, which will then be converted into a list of quoted identifiers and placed in the first slot of the loop object. The rule `in` detects when a valid request has been introduced by the user and, if the vending machine is idle, passes it as the next action to be attempted. For this example, 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 10.5. Recall that `BUYING-STRATS` includes the `META-LEVEL` module.

In the rule `in` below 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 10.4.1), and is placed in the action slot of the `State` triple; otherwise, an error message is placed in its output.

```
crl [in] :
    [QIL, < idle ; M ; nil >, QIL']
     => if T:ResultPair? :: ResultPair
        then [nil, < downTerm(getTerm(T:ResultPair?), idle) ; M ; nil >, QIL']
        else [nil, < idle ; M ; nil >, 'ERROR QIL]
        fi
     if QIL =/= nil
        /\ T:ResultPair? :=
             metaParse(upModule('VENDING-MACHINE-GRAMMAR, false), QIL, 'Action) .
```

For the other direction of the interaction, the rule `out` detects when the vending machine has a response to be output and, in that case, it places it in the output slot of the loop object.

```
crl [out] :
   [QIL, < A ; M ; QIL' >, QIL'']
   => [QIL, < A ; M ; nil >, QIL''  QIL']
   if QIL' =/= nil .
```

Next, we define the effects of the different actions in the state of the vending machine. The rules `showBasket` and `showCredit` extract the information about the remaining credit or the

items already bought, and place it in the output slot of the state; the rule `out` will then take care of moving it to the output slot of the loop object. 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 10.4.1).

```
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 [showBasket] :
    < showBasket ; M ; nil >
    => < idle ; M ; ('\u 'basket: '\o showBasket(M) '\n) > .

rl [showCredit] :
   < showCredit ; M ; nil >
   => < idle ; M ; ('\u 'credit: '\o showCredit(M) '\n) > .
```

The rules `insertCoin` 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 10.5) 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 [insertCoin] :
  < q ; M ; nil >
  => < idle ; downTerm(insertCoin('q.Coin, upTerm(M)), null) ; nil > .

rl [insertCoin] :
  < $ ; M ; nil >
  => < idle ; downTerm(insertCoin('$.Coin, upTerm(M)), null) ; nil > .
```

The rules `buy1Cake`, `buy1Apple`, and `buyNitems` implement the actions of buying one or more items. The strategy `onlyNitems` defined in the module `BUYING-STRATS` (see Section 10.5) 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 [buy1Cake]:
  < buy1Cake ; M ; nil >
  => < buy c : 1 ; M ; nil > .
```

```
  rl [buy1Apple]:
    < buy1Apple ;  M ; nil >
    => < buy a : 1 ; M ; nil > .

  rl [buyNitems]:
    < buy c : N ;  M ; nil >
    => < idle ; downTerm(onlyNitems(upTerm(M), 'buy-c, N), null) ; nil > .

  rl [buyNitems]:
    < buy a : N ;  M ; nil >
    => < idle ; downTerm(onlyNitems(upTerm(M), 'buy-a, N), null) ; nil > .
endm
```

11.3 Using the loop

We illustrate the basic ideas of using the loop with a sample session with the interface for the vending machine. Once the module `VENDING-MACHINE-INTERFACE` has been entered, we must first initialize the loop by setting its initial state by means of the `loop` command.

```
Maude> loop init .
```

We can see the state of the loop with the `continue` command, abbreviated `cont`, as follows:

```
Maude> cont .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result System: [nil,< idle ; null ; nil >,nil]
```

Once the loop has been initialized, we can input any data by writing it after the prompt enclosed in parentheses. For example,

```
Maude> ($)

Maude> (showCredit)
credit: $

Maude> ($)

Maude> cont .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result System: [nil,< idle ; $ $ ; nil >,nil]

Maude> (q)

Maude> (buy1Apple)

Maude> (showBasket)
basket: a

Maude> (showCredit)
credit: $ q q

Maude> ($)
```

```
Maude> (buy a : 3)

Maude> (showBasket)
basket: a a a a

Maude> (showCredit)
credit: q

Maude> cont .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result System: [nil,< idle ; q a a a a ; nil >,'\u 'credit: '\o '\r '\! 'q '\o '\n]
```

Note that, as already mentioned, the data in the output stream remains there after being printed; it is removed at the time of the next input event.

11.4 Metalanguage applications: tokens, bubbles, and metaparsing

The example in the previous sections is a toy example to illustrate the basic features of the module `LOOP-MODE`. However, the most interesting applications of this 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 handling the input/output and maintaining the persistent state of the object language interpreter or tool is done by `LOOP-MODE`. Full Maude (see Chapter 13) is entirely implemented in Maude using the methodology explained in this section.

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 data type $Sign_{\mathcal{L}}$ for $\mathcal{L}$-modules, and with auxiliary declarations for commands and other constructs. Maude provides great flexibility to do this, thanks to its mixfix front-end and to the use of *bubbles* (any nonempty 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: what we may call the *top-level* syntax of the language, and the user-definable syntax introduced in each module. The bubble data type allows us to reflect this duality of levels in the syntax definition. Similar ideas have been exploited using ASF+SDF [60, 61].

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. In fact, they can only be parsed with the grammar associated to 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 to 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 data types 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, nonempty 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))] .
```

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. We can, for example, declare the sort NeTokenList with constructor neTokenList as a list of identifiers, of any length greater 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       ( . ))] .
```

In general, to exclude identifiers I1, I2, ..., Ik, we use the syntax Exclude (I1 I2 ... Ik).

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
  including 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))] .

  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 .
  subsort Decl < DeclList .

  --- sort declaration
  op sort_. : Token -> Decl .

  --- operator declaration
  op op_:‘->_. : Token Token -> Decl .
  op op_:_->_. : Token NeTokenList Token -> Decl .

  --- equation declaration
  op eq_=_. : Bubble Bubble -> Decl .

  --- functional module
  op fmod_is_endfm : Token DeclList -> PreModule .
  op __ : DeclList DeclList -> DeclList [assoc gather(e E)] .
endfm
```

Notice how we explicitly declare operators that correspond to 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. The name of the sort `PreModule` reflects the fact that not all terms of this sort do actually represent 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.

As an example, we can call (in any module importing `META-LEVEL`) the operation `metaParse` with the metarepresentation of the module `MINI-MAUDE-SYNTAX` and the previous module `NAT3` transformed into a list of quoted identifiers.

```
Maude> red 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 .
```

that goes along the term metarepresenting the premodule looking for sort and operator declarations; these are obtained by means of auxiliary operations `extractSorts` and `extractOpDecls`, respectively. 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 .
```

Finally, the partial operation `processPreModule` 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 the following.

```
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 .

op processPreModule : Term ~> FModule .
eq processPreModule(T) = solveBubbles(T, extractSignature(T)) .

---- extractSignature
op extractSignature : Term ~> FModule .
op extractSorts : Term ~> SortSet .
op extractOpDecls : Term ~> OpDeclSet .

eq extractSignature('fmod_is_endfm['token[QI], T])
  = (fmod downTerm(QI, 'error) is
       nil
       sorts extractSorts(T) .
       none
       extractOpDecls(T)
       none
       none
     endfm) .

eq extractSorts('sort_.['token[T]]) = downTerm(T, 'error) .
eq extractSorts('__['sort_.['token[T1]], T2])
  = downTerm(T1, 'error) ; extractSorts(T2) .
ceq extractSorts(F[TL]) = none if F =/= '__ /\ F =/= 'sort_. .
ceq extractSorts('__[F[TL], T]) = extractSorts(T) if F =/= 'sort_. .

eq extractOpDecls('op_:_->_.['token[T1], 'neTokenList[TL], 'token[T2]])
  = (op downTerm(T1, 'error) : downTerm(TL, nil)
          -> downTerm(T2, 'error) [none] .) .
eq extractOpDecls('op_:`->_.['token[T1],'token[T2]])
  = (op downTerm(T1, 'error) : nil -> downTerm(T2, 'error) [none] .) .
ceq extractOpDecls(F[TL])
  = none
  if F =/= '__ /\ F =/= 'op_:_->_. /\ F =/= 'op_:`->_. .

eq extractOpDecls('__['op_:_->_.['token[T1],'neTokenList[TL],'token[T2]], T3])
  = (op downTerm(T1, 'error) : downTerm(TL, nil) -> downTerm(T2, 'error) [none] .
    extractOpDecls(T3)) .
eq extractOpDecls('__['op_:`->_.['token[T1],'token[T2]], T3])
  = (op downTerm(T1, 'error) : nil -> downTerm(T2, 'error) [none] .
     extractOpDecls(T3)) .
ceq extractOpDecls('__[F[TL], T2]) = extractOpDecls(T2)
  if F =/= 'op_:_->_. /\ F =/= 'op_:`->_. .

---- solveBubbles
op solveBubbles : Term FModule ~> FModule .
op solveBubblesAux : Term FModule ~> EquationSet .
op addEqs : FModule EquationSet -> FModule .

eq solveBubbles('fmod_is_endfm['token[QI], T], M)
  = addEqs(M, solveBubblesAux(T, M)) .
```

```
  eq solveBubblesAux('eq_=_.['bubble[T1], 'bubble[T2]], M)
    = (eq getTerm(metaParse(M, downTerm(T1, nil), anyType))
         = getTerm(metaParse(M, downTerm(T2, nil), anyType)) [none] .) .
  eq solveBubblesAux('__['eq_=_.['bubble[T1], 'bubble[T2]], T3], M)
    = (eq getTerm(metaParse(M, downTerm(T1, nil), anyType))
         = getTerm(metaParse(M, downTerm(T2, nil), anyType)) [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_=_. .

  eq addEqs(fmod QI is
              IL:ImportList
              sorts SS:SortSet .
              SubSorts:SubsortDeclSet
              OpDecls:OpDeclSet
              MembAxs:MembAxSet
              Eqs:EquationSet
            endfm,
            Eqs':EquationSet)
    = fmod QI is
        IL:ImportList
        sorts SS:SortSet .
        SubSorts:SubsortDeclSet
        OpDecls:OpDeclSet
        MembAxs:MembAxSet
        (Eqs:EquationSet Eqs':EquationSet)
      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))) .
rewrites: 33 in 1ms cpu (1ms real) (33000 rewrites/second)
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 :
            processPreModule(
              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))) .
rewrites: 49 in 0ms cpu (0ms real) (~ rewrites/second)
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 :
            processPreModule(
              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))) .
rewrites: 49 in 1ms cpu (4ms real) (49000 rewrites/second)
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
```

Chapter 12

Debugging and Troubleshooting

12.1 Debugging approaches

There are several approaches to debugging and optimizing Maude programs: tracing, term coloring, using the debugger, and using the profiler.

12.1.1 Tracing

The tracing facilities allow us to follow the execution of our specifications, that is, the sequence of rewrites or 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.

One of the more useful options is selective tracing:

```
set trace select on .
trace select foo bar ([_,_]) .
```

This will cause only rewrites where the statement 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 rewrites with redex whose top-level operator is `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 13). After loading Full Maude, its specification is excluded from the tracing, which allows us to trace Full Maude specifications as Core Maude specifications.

12.1.2 Term coloring

A common failure mode of Maude programs is when a term does not fully reduce. 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 non-constructors, that is, 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
  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 non-constructor 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.

12.1.3 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`, or hitting a break point.

Break points are set with the command

```
break select foo bar ([_,_]) .
```

where the names refer to operators or statement labels in a way that is completely analogous with the `trace select` command above. 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.

12.1.4 The profiler

Profiling is switched on by

```
set profile on .
```

When profiling is switched on, a count is kept of the number of executions of each statement. This can be displayed by

```
show profile  .
```

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

```
set clear profile on/off .
```

For unconditional statements, the profile information is just the number of rewrites using that statement. For conditional statements there is also the number of matches, since not every match leads to a rewrite due to condition failure. Also, when searching there can be multiple rewrites for each match, since the condition may be solved in multiple ways. Also, 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. Also, 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.

12.1.5 Performance note

Turning on tracing, break points or profiling causes Maude to run much more slowly, even if no additional output is produced. This is because these options perform extensive bookkeeping before and after each rewrite. To ensure Maude is running at full speed use:

```
set trace off .
set break off .
set profile off .
```

12.2 Traps and known problems

We list some commonly encountered problems with Maude.

12.2.1 Associativity and idempotence

Currently an operator cannot have both the `assoc` and `idem` attributes, as the necessary matching algorithms have not yet been implemented.

12.2.2 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
```

12.2.3 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.

12.2.4 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 FOO is
  sorts Foo Bar .
  op f : Foo -> Foo .
  op f : Foo -> Bar .
endfm
```

causes the following warning:

```
Warning: "overloading-assoc-warning.maude", line 4 (fmod FOO):
  declaration for f has the same domain kinds as the declaration on
  "overloading-assoc-warning.maude", line 3 (fmod FOO) 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 BAR is
  sort Foo .
  op f : Foo Foo -> Foo [assoc] .
  op f : Foo Foo Foo -> Foo .
endfm

Warning: "overloading-assoc-warning.maude", line 10 (fmod FOO):
  declaration for f clashes with declaration on "overloading-assoc-warning.maude",
  line 9 (fmod FOO) because of associativity.
```

12.2.5 Preregularity and associativity

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 also might be the kind, depending on the operator declarations.

For an operator that is declared associative (with the `assoc` attribute, see Section 4.4.1), there is an additional requirement, namely, 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 to 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 for binary operators whose arguments are not the same, but they 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 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 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 : f(a, a) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
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 if 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 type `[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 type `[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 ASSOCIATIVE1 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 ASSOCIATIVE1 : f(a, a) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result s1: a
```

or adding enough declarations to cover all possible combinations of arguments; in this case only one more is enough, as follows:

```
fmod ASSOCIATIVE2 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 ASSOCIATIVE2 : f(a, a) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result s1: a
```

Notice that when an associative operator is also declared to be commutative, Maude computes the commutative completion of the given sort declarations before checking preregularity and associativity.

12.2.6 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 FOO 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` $\leftarrow$ `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 NAT 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 FOO2 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 on; `FOO2` 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 meta-level 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.

12.2.7 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 BAR 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 15.3 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.

12.2.8 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 does not happen (for efficiency) 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-MB1 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-MB1 : f(e, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result [Foo]: f(e, a, b)

Maude> red f(a, b, e, a) .
reduce in ASSOC-MB1 : f(e, a, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result [Foo]: f(e, a, a, b)

Maude> red f(e, b, e, a) .
reduce in ASSOC-MB1 : f(e, e, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result [Foo]: f(e, e, a, b)

Maude> red f(a, b, e, e, a) .
reduce in ASSOC-MB1 : f(e, e, a, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result [Foo]: f(e, e, a, a, b)

Maude> red f(a, a, b, e, e, a) .
reduce in ASSOC-MB1 : f(e, e, a, a, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
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-MB1 above can be rewritten so that sort computations work as expected as follows:

```
fmod ASSOC-MB2 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-MB2 : f(e, a, b) .
rewrites: 2 in 0ms cpu (1ms real) (~ rewrites/second)
result Foo: f(e, a, b)

Maude> red f(a, b, e, a) .
reduce in ASSOC-MB2 : f(e, a, a, b) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Foo: f(e, a, a, b)

Maude> red f(e, b, e, a) .
reduce in ASSOC-MB2 : f(e, e, a, b) .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result [Foo]: f(e, e, a, b)

Maude> red f(a, b, e, e, a) .
reduce in ASSOC-MB2 : f(e, e, a, a, b) .
rewrites: 11 in 0ms cpu (0ms real) (~ rewrites/second)
result Foo: f(e, e, a, a, b)

Maude> red f(a, a, b, e, e, a) .
reduce in ASSOC-MB2 : f(e, e, a, a, a, b) .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Foo: f(e, e, a, a, a, b)
```

Here the declaration is at the kind level, and the effect of the declaration of f in ASSOC-MB1 is obtained by an extra membership axiom.[1]

Let us see another example of this situation, starting with a module specifying nonempty lists of natural numbers.

```
fmod 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 NAT-LIST and then defining a subsort of NatList.

[1] 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 as matching the extra membership essentially amounts to expanding out the equivalence class.

```
fmod SORTED-NAT-LIST is
  protecting 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 SORTED-NAT-LIST : 0 1 2 3 4 5 6 7 8 9 .
rewrites: 1354 in 0ms cpu (0ms real) (~ rewrites/second)
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-K is
  protecting NAT .
  sort NatList .
  subsort Nat < NatList .

  op __ : NatList NatList ~> NatList [assoc] .
  mb I:NatList J:NatList : NatList .
endfm

fmod SORTED-NAT-LIST-K is
  protecting NAT-LIST-K .
  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 .
reduce in SORTED-NAT-LIST-K : 0 1 2 3 4 5 6 7 8 9 .
rewrites: 1354 in 0ms cpu (0ms real) (~ rewrites/second)
result SortedNatList: 0 1 2 3 4 5 6 7 8 9
```

12.2.9 Memberships for iterated operators

Analogous 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. 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-MB1 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-MB1 : f(e) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Foo: f(e)

Maude> red f(f(e)) .
reduce in ITER-MB1 : f^2(e) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result [Foo]: f^2(e)

Maude> red f(f(f(e))) .
reduce in ITER-MB1 : f^3(e) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
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-MB2 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-MB2 : f(e) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Foo: f(e)

Maude> red f(f(e)) .
reduce in ITER-MB2 : f^2(e) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Foo: f^2(e)
```

```
Maude> red f(f(f(e))) .
reduce in ITER-MB2 : f^3(e) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Foo: f^3(e)
```

Here the 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.

Part II

Full Maude

Chapter 13

Full Maude Extensions

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 [27], and further documented in [28, 24, 25, 29]—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, parameterized views, 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 parameterized modules, views, and module expressions like summation, renaming and instantiation, were available in Full Maude long before they have been available in Core Maude [27]. In fact, Full Maude has not only been a complement to Core Maude, but also a place where to experiment with new 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, in the future, those features in Full Maude which are not available in Core Maude yet, will become part of it sooner or later.

In fact, Full Maude implements a complete user interface for the extended language. Using the `META-LEVEL` and `LOOP-MODE` 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 11. Thanks to the efficient implementation of the rewrite engine, the parser, and the module `META-LEVEL`, 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, being the main of these restrictions the fact that Full Maude inputs, including modules and commands, must be enclosed in parentheses, but there are others. These syntactic restrictions are explained in Section 13.5.

The structure of the chapter is as follows. Section 13.1 gives instructions on how to load and use Full Maude, how to enter modules, reduce terms, trace, etc. Section 13.2 explains how modules in Core Maude's database may be used in Full Maude. Section 13.3 introduces the additional module operations that are available in Full Maude. Section 13.4 explains how to move terms and modules up and down reflection levels. Finally, Section 13.5 enumerate the main differences between Full Maude 2.2 and Core Maude 2.2.

13.1 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 such specification, 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 2.2 '(December '1st', 2005')
```

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, one may also run Maude as follows:

```
> maude.linux full-maude.maude
                    \||||||||||||||||||/
                  --- Welcome to Maude ---
                    /||||||||||||||||||\
           Maude 2.2 built: Nov 28 2005 16:10:26
           Copyright 1997-2005 SRI International
                 Fri Dec 1 16:28:41 2005

           Full Maude 2.2 '(December '1st', 2005')
```

Note that the `full-maude.maude` file contains the executable specification of Full Maude. At the end of this file you can find the command

```
loop init .
```

which initializes the system just after loading the specification. This command starts an input-eval-loop (see Section 11.1) 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. `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 (which is quite likely), 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 15.11):

```
Maude> select FULL-MAUDE .
Maude> loop init .
```

In fact 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.

Let us recall from Section 11.1 that to get something into the `LOOP-MODE` system the text *has to be enclosed in parentheses*. This means that any module, theory, view, or command

intended for Full Maude has to be written enclosed in parentheses. Notice that, since Core Maude is still active—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. This may lead to some confusion, and we must 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 .
          eq source(f) = n2 .
          eq target(f) = n1 .
        endfm)
```

[1] Some fragments of this module have been discussed in Sections 3.5 and 4.3.

```
rewrites: 5786 in 141ms cpu (149ms real) (40752 rewrites/second)
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, but each of the modules or commands has to be enclosed in parentheses.

Note that when entering a module, as above, Maude gives us information on 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 of the cases it finally makes a call to Core Maude, Full Maude needs to do 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 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: 876 in 58ms cpu (66ms real) (14849 rewrites/second)
reduce in PATH :
  b ;(c ; d)
result Path :
  b ;(c ; d)

Maude> (red length(b ; (c ; d)) .)
rewrites: 391 in 4ms cpu (5ms real) (78215 rewrites/second)
reduce in PATH :
  length(b ;(c ; d))
result NzNat :
  3

Maude> (red a ; (b ; c) .)
rewrites: 344 in 4ms cpu (5ms real) (68813 rewrites/second)
reduce in PATH :
  a ;(b ; c)
result `[Path`] :
  a ;(b ; c)

Maude> (red source(a ; (b ; c)) .)
rewrites: 382 in 5ms cpu (5ms real) (63677 rewrites/second)
reduce in PATH :
  source(a ;(b ; c))
result `[Node`] :
  source(a ;(b ; c))

Maude> (red target((a ; b) ; c) .)
rewrites: 388 in 5ms cpu (5ms real) (64677 rewrites/second)
reduce in PATH :
  target((a ; b); c)
result `[Node`] :
  target((a ; b); c)
```

```
Maude> (red length(a ; (b ; c)) .)
rewrites: 382 in 5ms cpu (5ms real) (63677 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) (~ rewrites/second)
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) (~ rewrites/second)
result NzNat: 3
```

Tracing, debugging, profiling, and the other facilities available in Core Maude are also available for Full Maude. It is Core Maude the one providing these facilities, and therefore the commands for managing them must be given to it 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) .)
********** equation
eq target(b) = n3 . target(b)
--->
n3
********** equation
eq source(c) = n3 . source(c)
--->
n3
********** 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: 325 in 6ms cpu (8ms real) (46435 rewrites/second)
reduce in PATH :
  length(b ; c)
result NzNat :
  2
```

One must always have 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 12.1.1 and 15.5). 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.

13.2 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. 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 database, it looks into Core Maude's one.

When metaprogramming, the system behaves differently. In Core Maude, a module at the metalevel can include a module at the object level. In Full Maude, however, modules at the metalevel 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 input-eval-print loop (see Section 11.1). Assuming there is a file `path.maude` containing the Core Maude module `PATH`, we will have the following.

```
> maude.linux full-maude.maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.2 built: Nov 28 2005 16:10:26
            Copyright 1997-2005 SRI International
                  Fri Dec 1 16:28:41 2005

            Full Maude 2.2 `(December `1st`, 2005`)

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 2.2 `(December `1st`, 2005`)

Maude> (red in PATH : b ; (c ; d) .)
rewrites: 3718 in 177ms cpu (181ms real) (20890 rewrites/second)
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 module entered before the reinitialization will have to be reentered. 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, it 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 modules which are not part of Maude's prelude. For example, for using 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 model-checker.maude full-maude.maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.2 built: Nov 28 2005 16:10:26
            Copyright 1997-2005 SRI International
                  Fri Dec 1 16:28:41 2005

            Full Maude 2.2 `(December `1st`, 2005`)

Maude> (mod CHECK-RESP is
          protecting MODEL-CHECKER .
          ...
        endm)

Maude> (red p(0) |= (<> Qstate) .)
```

See Section 14.7 for a concrete example of the use of Maude's model checker on Full Maude modules.

13.3 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 modules (see Chapter 6). All the predefined modules introduced in Chapter 7, plus the module `META-LEVEL` and its submodules, described in Chapter 10, are also available in Full Maude.[2]

In addition to the module operations available in Core Maude, Full Maude supports the following extensions:

- *tuple expressions* which, given any nonzero natural number, generate parameterized modules specifying *tuples* of the corresponding size.

  `TUPLE [` ⟨*NonzeroNaturalNumber*⟩ `]` {⟨*ViewExpression*⟩}

  See Section 13.3.1.

- *parameterized views*, and the instantiation of parameterized modules with instantiations of views. See Section 13.3.2.

- *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, being also possible the instantiation of such object-oriented parameterized modules. We devote Chapter 14 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. In the case of Maude, this flattened module will then be compiled into the 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 rewrite rules that operate on the metalevel term representations of modules. This is essentially the idea behind the implementation of Full Maude in Maude.

The use of module expressions is in Full Maude a bit more general than in Core Maude. In Full Maude a 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 (if the top level is in fact a module; note that if the top level is a theory then it must be imported in `including` mode),

- the source or target of a view, or

- the parameter of a module, provided the top level is a theory.

But 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
```

[2]The predefined module `LOOP-MODE` described in Section 11.1 is not supported in Full Maude.

```
Maude> (show ops LISTNat .)
  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 does not work only with predefined modules. Let us do the same with the instantiation of the `MAX` module presented in Section 6.3.4 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 Core Maude views from Full Maude. Any 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 `MAX{IntAsToset}` requires view `TOSET` and `IntAsToset`.

```
Maude> (red in MAX{IntAsToset} : max((5, 4, 8, 4, 6, 5)) .)
result NzNat :
  8
```

Similarly, 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.

The summation module expression is also available in Full Maude. To illustrate its use, let us consider, for example, the theory of semirings given in Section 6.3.1.

```
(fth SEMIRING2 is
   including MONOID + +MONOID .
   vars X Y Z : Elt .
   eq X (Y + Z) = (X Y) + (X Z) [nonexec] .
   eq (X + Y) Z = (X Z) + (Y Z) [nonexec] .
   eq 1 X = X .
endfth)
```

13.3.1 The n-tuple module expression

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]` produces as its 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)
```

The Clear/OBJ module operations take theories, modules, and views, and return new theories and modules; 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 the n-tuple module expression is in principle of a completely different nature from the usual Clear/OBJ module operations, the way of handling it is the same as the way of handling 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 have the following specification.

```
(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)
```

13.3.2 Parameterized views

Suppose we have already 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 view `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 built-in 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 FOO{X :: TRIV} is
   protecting LIST{Set{X}} .
 endfm)
```

Instantiating the module FOO with a view V from TRIV to another module or theory results in a module with name FOO{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. 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. Of course, for such a set to represent a function there cannot be two pairs associating different results to 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 cannot be given on associative operators over sorts (see Section 12.2.8), we cannot use either the specification of sets described in Section 6.3.3 or the predefined module in Section 7.11.2. Let us consider instead the following module:[3]

```
(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} .

  var E : X$Elt .

  *** idempotence
  eq E, E = E .
 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 TUPLE[2]{X, Y} defining pairs of elements. The appropriate parameterized view can be defined as follows:

[3]Note the use of the equivalent single-identifier-form for on-the-fly declarations of variables; as we will see in Section 13.5, this is one of the syntactic restrictions of Full Maude.

```
(view Tuple{X :: TRIV, Y :: TRIV} from TRIV to TUPLE[2]{X, Y} is
   sort Elt to Tuple{X, Y} .
 endv)
```

The partiality of a function can be specified by defining a default value to be used as the result for the input elements for which the function is not defined. We define the parameterized functional module `DEFAULT-ELEMENT{X :: TRIV}` in which we declare a sort `Default{X}` as a supersort of the sort `Elt` of the parameter theory, and a constant `null` of sort `Default{X}`.

```
(fmod DEFAULT-ELEMENT{X :: TRIV} is
   sort Default{X} .
   subsort X$Elt < Default{X} .
   op null : -> Default{X} .
 endfm)
```

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 module expression `SET-KIND{Tuple{X, Y}}` with sorts `KSet{Tuple{X, Y}}` and `NeKSet{Tuple{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{Tuple{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 Core Maude predefined module `SET` (see Section 7.11.2) 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{Tuple{X, Y}} .
   pr SET{X} + SET{Y} .
   pr DEFAULT-ELEMENT{Y} .

   sort PFun{X, Y} .
   subsorts Tuple{X, Y} < PFun{X, Y} < KSet{Tuple{X, Y}} .

   vars A D : X$Elt .
   vars B C : Y$Elt .
   var  F : PFun{X, Y} .
   var  S : KSet{Tuple{X, Y}} .

   op dom : KSet{Tuple{X, Y}} -> Set{X} .          *** domain
   eq dom(empty) = empty .
   eq dom((A, B), S) = A, dom(S) .
   op im : KSet{Tuple{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 -> Default{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 ] = null [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, the view `Nat`, in order to get the finite partial functions from natural numbers to natural numbers by means of the module expression `PFUN{Nat, Nat}`.

13.4 Moving up and down between reflection levels

The functions provided by Core Maude for moving up reflection levels (see Section 10.4.1) are not very useful in Full Maude. Although they are available as part of the module `META-LEVEL`, they take as one of its 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. However, 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 MY-NAT 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]]`.

13.4.1 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;[4] `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 `MY-NAT` has been entered into Full Maude, and therefore it is in its database, we can get its metarepresentation, which we denoted by $\overline{\texttt{MY-NAT}}$, as follows:

[4] The Core Maude `upModule` function takes as second argument a Boolean value (see Section 10.4.1).

```
Maude> (red in META-LEVEL : upModule(MY-NAT) .)
result FModule :
  fmod 'MY-NAT 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 10.3) to `upModule(MY-NAT)` as follows:

```
Maude> (red in META-LEVEL : getOps(upModule(MY-NAT)) .)
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 the descent functions as discussed in Section 10.4.

```
Maude> (red in META-LEVEL :
          metaReduce(upModule(MY-NAT), '_+_['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 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(MY-NAT), upTerm(MY-NAT, 0 + s 0)) .)
result ResultPair :
  {'s_['0.Nat],'Nat}
```

As another example, to obtain the metarepresentation of the term `s 0` in the module `MY-NAT` above, which we denoted by $\overline{\texttt{s 0}}$, we can write

```
Maude> (red in META-LEVEL : upTerm(MY-NAT, s 0) .)
result GroundTerm :
  's_['0.Nat]
```

Note that 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, in a module importing the module `META-LEVEL`.

13.4.2 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 term from its metarepresentation. The `down` command takes two arguments. The first argument is the name of the module to which the term to be returned belongs. The metarepresentation of the desired output term should be the result of the command given as second argument. The syntax of the `down` command is as follows:

`down` ⟨*ModuleExpression*⟩ `:` ⟨*Command*⟩

Thus, we can give the following command.

```
Maude> (down MY-NAT :
          red in META-LEVEL :
            getTerm(metaReduce(upModule(MY-NAT), upTerm(MY-NAT, 0 + s 0))) .)
result Nat :
  s 0
```

Notice that this is equivalent to what we may write using the overline notation as:

```
Maude> red metaReduce(MY-NAT, s 0 + 0) .
result Term: 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 getTerm(
              metaReduce(upModule(META-LEVEL),
                upTerm(META-LEVEL,
                  getTerm(
                    metaReduce(upModule(MY-NAT), upTerm(MY-NAT, 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 metaReduce(META-MY-NAT, metaReduce(MY-NAT, s 0 + 0)) .
result Term: s 0
```

Of course, we can mix up `down` and `upModule` and `upTerm`:

```
Maude> (down MY-NAT :
          down META-LEVEL :
            red getTerm(
                  metaReduce(upModule(META-LEVEL),
                    upTerm(META-LEVEL,
                      getTerm(
                        metaReduce(upModule(MY-NAT), upTerm(MY-NAT, 0 + s 0)))))) .)
result Nat :
  s 0
```

The metalevel function `downTerm` can also be used, although note that it is a Core Maude function, and therefore can only be used on Core Maude modules.

```
Maude> (down MY-NAT :
          red in META-LEVEL :
            downTerm(
              getTerm(
                metaReduce(upModule(META-LEVEL),
                  upTerm(META-LEVEL,
                    getTerm(
                      metaReduce(upModule(MY-NAT), upTerm(MY-NAT, 0 + s 0))))),
              'T:Term) .)
rewrites: 37193 in 293ms cpu (307ms real) (126525 rewrites/second)
result Nat :
  s 0
```

13.5 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 Section 13.3 and later in Chapter 14), we find a number of differences between Full Maude and Core Maude. There are some obvious ones, like the fact that any module, theory, view or command entered into Full Maude must go enclosed in parentheses, or the differences in printing, tracing, debugging, etc., but there are also others that impose certain limitations on the specifications themselves:

1. Operator and message names have to be given in their equivalent *single identifier form* when they are declared (but they can later be written in the usual way in statements and in terms for evaluation).

2. Sort names used in term qualifications, membership assertions, and on-the-fly declarations of variables have to be in their equivalent *single identifier form*.

3. The `continue`, `show component`, `show path`, and `show search graph` commands are not supported in Full Maude.

In the rest of the section we explain the first two restrictions in some detail and give some hints on how to avoid them.

Operator names have to be given as a single identifier. To declare multi-identifier operators they have to be given 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 for Core Maude. For example, the declaration of this operator on sort, say, `Int` is as follows.

```
op _less‘than‘or‘equal_ : Int Int -> Bool .
```

Notice 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 write

```
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 .
```

Notice that, 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` 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 has to do with the qualification of terms by sort names, with on-the-fly declarations of variables, and with 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` being declared on the fly of sort `List{Nat}` should be written `L:List‘{Nat‘}`. Similarly, to check whether a term `T` has the sort `List{Nat}` we have to write `T : List‘{Nat‘}` or `T :: List‘{Nat‘}`.

Chapter 14

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. In particular, all object-oriented modules implicitly include the `CONFIGURATION+` module below, which in turn includes the module `CONFIGURATION` (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, it includes the declarations of sorts

- `Oid` of object identifiers,
- `Cid` of class identifiers,
- `Object` for objects, and
- `Msg` for messages.

The module `CONFIGURATION+` imports the module `CONFIGURATION`, defines a function `class` which returns the actual class of the given object, and adds syntax for objects with no attributes `< _:_| >`.

```
mod 'CONFIGURATION+ is
  including 'CONFIGURATION .
  op <_:_| > : Oid Cid -> Object .
  op class : Object -> Cid .
  eq < O:Oid : C:Cid | > = < O:Oid : C:Cid | none > .
  eq class(< O:Oid : C:Cid | A >) = C:Cid .
endm
```

As in Core Maude, we may have specifications of object-oriented systems in system modules; for example, we could enter the system modules describing object-based systems discussed in Chapter 8. However, although Maude's system modules are sufficient for specifying object-oriented systems, there are of course important conceptual advantages provided by the syntax of object-oriented modules. It 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 were provided.

Object-oriented modules are however just syntactic sugar: they are internally transformed into system modules for execution purposes (Section 14.8). See [24] 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.

14.1 Object-oriented systems

Some of the concepts related with object orientation were introduced in Chapter 8. Here we recall some of them and then focus on the notions of class and inheritance, and on the syntactic facilities provided by Full Maude to support object-oriented programming.

14.1.1 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 '`:`', 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. Such syntactic form can be 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.

The concurrent state of an object-oriented system is then a multiset of objects and messages, called a `Configuration`, with multiset union described with empty syntax `__`, and with `assoc`, `comm`, and `id: none` as operator attributes.

14.1.2 Classes

Classes are defined with the keyword `class`, followed by the name of the class, and 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 sorts, and in particular, class names may be parameterized in a way completely similar to parameterized sort names (see Section 6.3.3).

For example, the class `Accnt` of bank account objects introduced in Section 8.1 may be declared as follows:

```
class Accnt | bal : Int .
```

With this syntax, declarations for the class identifier `Accnt` and attribute `bal` are implicit.

A class `Person`, with a name, an age, and a bank account 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`.

In Full Maude object-oriented modules there is an operation `class` that takes an object as argument and returns its actual class. Thus, for example,

```
class(< A-002 : Accnt | bal : 1000 >)
```

returns the class identifier `Accnt`. This operation will be particularly useful when combined with the inheritance relation (see the use of the `class` operation in the example in Section 14.5).

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 multiple message operator declarations. The possibility of declaring messages of different types may be useful, i.e., for restricting the kind of messages that could be consumed by a particular type of object. As in the case of operators, messages 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`, or the objects involved in a message, the source and the target accounts of an account transfer in the `from_to_transfer_` message. Note also that, as explained in Chapter 8, Maude 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 specified by rewrite rules in a declarative way (see Section 14.1.4).

Given object identifiers `Peter` and `A-002`, the following term may represent a configuration with a person, his/her account, and a `credit` message sent to it.

```
< Peter : Person | name : "John", age : 34, account : A-002 >
< A-002 : Accnt | bal : 1000 >
credit(A-002, 100)
```

14.1.3 Inheritance

Class inheritance is directly supported by Maude's order-sorted type structure. 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 as well as 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 `SAV-ACCNT` of saving accounts introducing a subclass `SavAccnt` of `Accnt` with a new attribute `rate` recording the interest rate of the account.

```
class SavAccnt | rate : Float .
subclass SavAccnt < Accnt .
```

In this example there is only one class immediately above `SavAccnt`, namely, `Accnt`. 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 to it in each of those superclasses must be the same. 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 `SavAccnt` will have an attribute `bal` and can receive messages debiting, crediting and transferring funds exactly as any other object in the class `Accnt`. For example, the following object is a valid instance of class `SavAccnt`.

```
< A-002 : SavAccnt | 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 such as

```
subclasses A B C < D E < F G H .
```

14.1.4 Object-oriented rules

The behavior associated with the messages is specified by rewrite rules in a declarative way. For example, the semantics of the messages `credit`, `debit`, and `from_to_transfer_` declared in Section 14.1.2 may be given as follows:

```
vars A B : Oid .
var  M : Nat .
vars N N' : Int .

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

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

crl [transfer] :
  (from A to B transfer M)
  < A : Accnt | bal : N >
  < B : Accnt | bal : N' >
  => < A : Accnt | bal : N - M >
     < B : Accnt | 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 [46].

We simplify the notation used in object-oriented modules by giving the user the possibility of not mentioning 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 righthand side do not matter, and all attributes not explicitly mentioned are left unchanged (see [46, 24] for details).

For instance, a message for changing the age of a person defined by the class `Person` (introduced in Section 14.1.2) 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 > .
```

Attributes `name` and `account` not mentioned in this rule are not changed by the rule. The value of the attribute `age` is replaced by the given new age, independently of its previous value.

The following module `ACCNT` contains all the declarations above defining the class `Accnt`. Note that `Qid` is made a subsort of `Oid`, making any quoted identifier a valid object identifier.

```
(omod ACCNT is
   protecting QID .
   protecting INT .

   subsort Qid < Oid .
   class Accnt | 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 : Accnt | bal : N >
     => < A : Accnt | bal : N + M > .

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

   crl [transfer] :
     (from A to B transfer M)
     < A : Accnt | bal : N >
     < B : Accnt | bal : N' >
     => < A : Accnt | bal : N - M >
```

```
        < B : Accnt | 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 : Accnt | bal : 2000 >
            debit('A-06238, 1000) .)

result Object : < 'A-06238 : Accnt | bal : 1000 >
```

The following module contains the declarations for the class `SavAccnt`.

```
(omod SAV-ACCNT is
   including ACCNT .
   protecting FLOAT .
   class SavAccnt | rate : Float .
   subclass SavAccnt < Accnt .
 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[1] attribute in account objects.

We can now rewrite a configuration, obtaining the following result.

```
Maude> (rew < 'A-73728 : SavAccnt | bal : 5000, rate : 3.0 >
            < 'A-06238 : Accnt | bal : 2000 >
            < 'A-28381 : SavAccnt | bal : 9000, rate : 3.0 >
            debit('A-06238, 1000)
            credit('A-73728, 1300)
            credit('A-28381, 200) .)

result Configuration :
            < 'A-06238 : Accnt | bal : 1000 >
            < 'A-73728 : SavAccnt | bal : 6300, rate : 3.0 >
            < 'A-28381 : SavAccnt | bal : 9200, rate : 3.0 >
```

14.2 Example: a rent-a-car store

In order to further illustrate the specification of object-oriented systems we will specify in Maude a simple example, a rental car store. The regulations of the system, especially those that rule the rental processes, are:

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.

[1] See [52] for a general methodology to specify real-time systems in rewriting logic.

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% 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., its structure. We can identify three main classes, namely the store, customers, and cars. There are three special kinds of customers (staff, casual customers, and regular customers), and 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 additional attributes. The system also requires some control over time, which we get with a class representing calendars which provides the current date and simulates the passage of time.

The `Customer` class has three attributes, namely `cash`, `debt`, and `suspended` for keeping record, respectively, of the amount of cash that the customer currently has, his debt with the store, and whether he is suspended or not. Such a class may be defined by the following Maude declaration:

```
class Customer | cash : Nat, debt : Nat, suspended : Bool .
```

The attribute `available` of the class `Car` 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, nor with any other issues related to the specification of time. See the paper by Ölveczky and Meseguer [52] on the specification of real-time systems in rewriting logic and Maude for a discussion on these issues.

Four actions can be identified in the example:

1. a customer rents a car,
2. a customer returns a rented car,
3. a customer is suspended for being late in paying her debt or for being late in returning a rented car, and
4. a customer pays (part of) her debt.

The rental of a car by a customer is specified by the rule `car-rental` 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 those attributes of an object that are not relevant for an axiom do not need to be mentioned. Attributes not appearing in the righthand side of a rule will maintain their previous values unmodified. Note that the variables `A` and `NumDays` appear in the righthand side or condition of the rule but not in its lefthand side (this is the reason why this rule is declared to be `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 at that time. 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, 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 part of the amount due not covered by the deposit.

Debts may be satisfied at any time, the only condition being that the amount paid is between zero and the amount of money owned 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 10.5); a concrete possibility will be described later in Section 14.6.

14.3 Object-oriented parameterized programming

The notions of theory, view, and parameterized module have been extended to the object-oriented case. In this section we explain how we can write object-oriented theories, views with object-oriented theories or modules as sources or targets, and object-oriented parameterized modules with possibly object-oriented theories as parameters. We will see in Section 14.4 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 with an object-oriented theory with a view with another object-oriented module as target.

14.3.1 Theories

In addition to functional and system theories, Full Maude also supports object-oriented theories. Their structure is the same as that of their module counterparts. 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)
```

14.3.2 Views

For views with object-oriented theories as 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 to the declaration of the given message.

14.3.3 Parameterized modules

Like any other type of module, object-oriented modules can be parameterized, and like sorts class names may be parameterized. The naming of parameterized classes follows the same conventions discussed in Section 6.3.3 for parameterized sorts.

As an example of 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 attribute `first` is `null`. Each object of class `Node{X}` has an attribute `next` holding the identifier of the next node—which will 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 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 nonempty stack
      < O : Stack{X} | first : O' >
      < O' : Node{X} | contents : E >
      (O top O'')
      => < O : Stack{X} | >
         < O' : Node{X} | >
         (O'' elt E) .

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

   rl [push2] : *** push on an empty stack
      < O : Stack{X} | first : null >
      (O push E)
```

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

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

Notice that `top` and `pop` messages are not consumed if the stack is empty.

We may want to define stacks not storing 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 `STACK2` below, which is parameterized by the object-oriented theory `CELL` introduced in Section 14.3.1. 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 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 nonempty 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 nonempty stack
      < O : Stack{X} | first : o(O, N) >
      (O push O')
      => < O : Stack{X} | first : o(O, N + 1) >
         < o(O, N + 1) : Node{X} |
                next : o(O, N), node : O' > .
```

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

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

14.4 Module operations on object-oriented modules

The Full Maude module operations—summation, renaming, and instantiation—have been extended so that they are also available on object-oriented modules.

14.4.1 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. Thus, in addition to the renamings available in Core Maude, Full Maude also supports 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.

```
Maude> (show module 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 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+ .
  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$Elt -> Msg .
   rl < O:Oid : Stack{X}| head : O':Oid >
      < O':Oid : Node{X}| next : O'':Oid >
      O:Oid pop
      => < O:Oid : Stack{X}| head : O'':Oid >
      [label pop] .
   rl < O:Oid : Stack{X}| head : O':Oid >
      < O':Oid : Node{X}| node : O'':Oid >
      < O'':Oid : X$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
```

In the current version, Full Maude does not canonize type arguments in operator renamings as Core Maude does (see Section 6.2.2).

14.4.2 Module instantiation

We show in this section how by instantiating the object-oriented module `STACK2` given in Section 14.3.3 we can obtain a specification of a stack of banking accounts. We first specify a view `Accnt` from the object-oriented theory `CELL` (in Section 14.3.1) to the object-oriented module `ACCNT` (in Section 14.1.4).

```
(view Accnt from CELL to ACCNT is
   sort Elt to Int .
   class Cell to Accnt .
   attr Cell . contents to bal .
endv)
```

Now we can do the following rewriting on the module resulting from the instantiation.

```
Maude> (rew in STACK2{Accnt} * (class Accnt to Account,
                                class Stack{Accnt} to Stack{Account},
                                class Node{Accnt} to Node{Account},
                                attr Stack{Accnt} . first to head,
                                attr Accnt . 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 >
```

14.5 Example: extended rent-a-car store

This section describes a variation of the rent-a-car store example in Section 14.2, in which several interesting data structures are used to store relevant information.

Let us reconsider the specification of a rent-a-car store presented in Section 14.2 by adding the following regulations:

1. When a rented car is returned, the deposit is used to pay the rental charges, which are calculated in accordance to the conditions at pick-up time.
2. Staff members can also rent cars.
3. Staff members and preferred customers benefit from special discounts in all rentals.
4. 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. Note that we need to distinguish now three different types of customers, namely casual customers, preferred customers, and staff members, being possible for a casual customer to be promoted to preferred if he spends a given amount of money.

As an alternative approach, 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 casual 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. Please note that 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}, Int},
  payments  : PFun{Oid, Int},
  penalty   : Int,
  threshold : Int,
  suspended : Set{Oid},
  rates     : PFun{Cid, Int},
  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 (see Section 13.3.2), 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, Int}`, 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, Int}`, 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` 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}`. Also, notice the use of the predefined sorts `Oid` and `Cid` 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 : Int, debt : Int .
class Staff .
class CasualCust .
class PreferredCust .
subclasses CasualCust PreferredCust Staff < Customer .

class Car | available : Bool .
class EconomyCar .
class MidSizeCar .
class FullSizeCar .
subclasses EconomyCar MidSizeCar FullSizeCar < Car .

class Rental |
  deposit    : Int,  discount   : Int,
  dueDate    : Int,  pickUpDate : Int,
  rate       : Int,  customer   : Oid,
  car        : Oid .
```

Note that the definition of classes `Customer`, `Car` (and its subclasses), and `Rental` are the same as those in the example in Section 14.2.

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 : M + - Amnt >
     < I : Car | available : false >
     < V : Store | rentals : (A, RS) >
     < C : Calendar | >
     < A : Rental | pickUpDate : Today, dueDate : Today + NumDays, car : I,
       deposit : Amnt, customer : U, rate : Rt, discount : Dscnt >
  if not U in US                          *** the customer is not suspended
     /\ Rt := Rts[class(< I : Car | >)]
     /\ Dscnt := Dscnts[(class(< U : Customer | >), class(< I : Car | >))]
     /\ Amnt := (Rt + - Dscnt) * NumDays
     /\ M >= Amnt                         *** enough cash to make a deposit
  [nonexec] .
```

Note the use of 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. 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 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. Note that the function `class` takes an object as argument and returns its actual class (see Section 14.1.2). Thus, if the `Car` object to which the above rule applies is, for instance, `< 'c123 : MidSizeCar | ... >`, then the `class` function applied to it returns `MidSizeCar`, and not `Car`. Finally, note 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 + Dpst + - Amnt >
     < I : Car | available : true >
     < V : Store | rentals : RS,
       payments : (if Pmnts[U] == null     *** no record for this customer
                   then Pmnts[U -> Amnt]
                   else Pmnts[U -> ((Pmnts[U]) + Amnt)]
                   fi) >
     < C : Calendar | >
  if (Today <= Ddt) /\ Amnt := (Rt + - Dscnt) * (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 + Amnt + - Dpst >
     < I : Car | available : true >
     < V : Store | rentals : RS,
       payments : (if Pmnts[U] == null
                   then Pmnts[U -> Dpst]
                   else Pmnts[U -> ((Pmnts[U]) + Dpst)]
                   fi) >
     < C : Calendar | >
  if Ddt < Today                          *** it is returned late
     /\ Amnt := ((Rt + - Dscnt) * (Ddt + - Ppdt))
           + ((((Rt + - Dscnt) * (Today + - Ddt)) * (100 + Pnlt)) quo 100) .
```

In this case the customer's debt is increased by the part of the amount due not covered by the deposit.

Debts may be satisfied 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 : M + - Amnt, cash : 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 most cases 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 : CasualCust | 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 an object of class `CasualCust` becomes of class `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 will see an example of such a strategy in the next section.

14.6 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 14.2 and 14.5. Dealing with strategies may become cumbersome, since we need to handle terms and modules at different levels of reflection, and things may become really hard to read and handle. We also show in this section how the `upModule` and `upTerm` functions and the `down` command introduced in Section 13.4 may help.

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 13.3.1. Thus, to get the module expression `LIST{Tuple{Qid, Substitution}}`, and given the predefined view `Qid` and the *parameterized view* `Tuple`, that we have already used in the partial functions example of Section 13.3.2, 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 partial function `rewSeq` in this module takes the metarepresentation of a module, the metarepresentation of a term, and a list of pairs (rule label - substitution), and applies the given rules sequentially, using in their applications their corresponding partial substitutions.

```
(fmod REW-SEQ is
   inc META-LEVEL .
   pr 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 .
 endfm)
```

The rules to be applied here are part of the module given as first argument. The function 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 10.4.3). 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 will illustrate the use of this strategy by applying a sequence of rules on a configuration of the rent-a-car system specified in Section 14.2. Let `RENT-A-CAR-STORE` be the name of the module containing the specification of such a system, and let `StoreConf` be the following configuration of objects.

```
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 > .
```

This configuration consists of two clients `C1` and `C2`, three cars `A1`, `A3` and `A5`, and a calendar object `C`. Now, let `StoreStrat` be the following sequence of pairs (rule label - substitution) that defines the strategy—by means of a sequence of actions in this case:

```
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]) .
```

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),
                     upTerm(RENT-A-CAR-STORE, 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 14.5, contained in a module named `EXTENDED-RENT-A-CAR-STORE`, as follows.

```
Maude> (down EXTENDED-RENT-A-CAR-STORE :
          rew rewSeq(upModule(EXTENDED-RENT-A-CAR-STORE),
                ---- initial configuration
                upTerm(EXTENDED-RENT-A-CAR-STORE,
                  < 'S : Store |
                       discounts : (((Staff,EconomyCar),20)
                                    ((Staff,MidSizeCar),30)
                                    ((Staff,FullSizeCar),40)
                                    ((CasualCust,EconomyCar),0)
                                    ((CasualCust,MidSizeCar),0)
                                    ((CasualCust,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 : CasualCust | cash : 5000, debt : 0 >
                  < 'A1 : EconomyCar | available : true >
                  < 'A3 : MidSizeCar | available : true >
                  < 'A5 : FullSizeCar | available : true >
                  < 'C : Calendar | date : 0 >),
                ---- strategy
                ('car-rental,                      *** client C1 rents the mid
                   '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])) .)
```

The resulting state shows how, after eight days, client `C1` has paid $500, and has a debt of $60.

```
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 : CasualCust | cash : 5000,debt : 0 >
  < 'S : Store | calendar : 'C,
                 cars :('A1, 'A3, 'A5),
                 customers :('C1, 'C2),
                 discounts :(((CasualCust,EconomyCar),0)
                             ((CasualCust,FullSizeCar),0)
                             ((CasualCust,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 >
```

14.7 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 on it. Both the algorithm and the property guaranteing that all processes reenter their critical sections are parameterized with respect to the number of processes. We use Maude's model checker to prove the mutual exclusion and guaranteed reentrance properties. As we said in Section 13.2, 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 `NAT*`, which requires a constant of sort `Nat`. Thus, having a view, say `5` from `TRIV` to `NAT*` mapping `*` to the natural number `5`, the module expression `NAT/{5}` specifies the natural numbers modulo five.

```
(fth NAT* is
   protecting NAT .
   op * : -> Nat .
 endfth)

(fmod NAT/{N :: NAT*} is
```

```
    sort Nat/{N} .
    op `[_`] : Nat -> Nat/{N} .
    op _+_ : Nat/{N} Nat/{N} -> Nat/{N} .
    op _*_ : Nat/{N} Nat/{N} -> Nat/{N} .
    vars N M : Nat .
    eq [*] = [0] .
    ceq [s(N)] = [s(M)] if [M] := [N] /\ M < 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 :: NAT*} is
  protecting NAT/{N} .

  sort Mode .
  ops wait critical : -> Mode .

  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 9 for a discussion on the use

of Maude's model checker). `inCrit` takes a `Nat/{N}` as argument, thus making this property parameterized with respect to the number of processes, and is true when such a process is in its critical section. `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`, 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 :: NAT*} is
  pr RROBIN{N} .
  inc MODEL-CHECKER .
  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)
```

We now prove mutual exclusion and guaranteed reentrance for five processes.

```
(view 5 from NAT* to NAT is
   op * to term 5 .
 endv)

Maude> (red in CHECK-RROBIN{5} : modelCheck(init, [] ~ twoInCrit) .)
result Bool :
  true
```

```
Maude> (red in CHECK-RROBIN{5} : modelCheck(init, [] guaranteedReentrance) .)
result Bool :
  true
```

Of course the answer 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})
```

14.8 From object-oriented modules to system modules

The best way to understand classes and class inheritance in 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 Maude's object-oriented modules provide convenient syntax for programming object-oriented systems, their semantics can be reduced to that of system modules.[2] 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 `ACCNT` module introduced earlier. Note that a subsort `Accnt` of `Cid` is introduced. The purpose of this subsort is to range over the class identifiers of the subclasses of `Accnt`. For the moment, no such subclasses have been introduced; therefore, at present the only constant of sort `Accnt` is the class identifier `Accnt`.

```
mod ACCNT is
  including INT .
  including QID .
  including CONFIGURATION+ .
  sorts Accnt .
  subsort Qid < Oid .
  subsort Accnt < Cid .
  op Accnt : -> Accnt .
  op credit : Oid Int -> Msg .
  op debit : Oid Int -> Msg .
  op from_to_transfer_ : Oid Oid Int -> Msg .
  op bal :_ : Int -> Attribute .
  var A : Oid .
  var B : Oid .
  var M : Int .
  var N : Int .
  var N' : Int .
  var V@Accnt : Accnt .
  var ATTS@0 : AttributeSet .
  var V@Accnt1 : Accnt .
  var ATTS@2 : AttributeSet .
  rl [credit] :
    credit(A, M)
    < A : V@Accnt | bal : N, ATTS@0 >
```

[2]The underlying system modules still have the special features supported in Core Maude for object-based programming as explained in Chapter 8.

```
    => < A : V@Accnt | bal : (N + M), ATTS@0 > .
  crl [debit] :
    debit(A, M)
    < A : V@Accnt | bal : N, ATTS@0 >
    => < A : V@Accnt | bal : (N - M), ATTS@0 >
    if N >= M = true .
  crl [transfer] :
    (from A to B transfer M)
    < A : V@Accnt | bal : N, ATTS@0 >
    < B : V@Accnt1 | bal : N', ATTS@2 >
    => < A : V@Accnt | bal : (N - M), ATTS@0 >
       < B : V@Accnt1 | 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:[3]

- The module `CONFIGURATION+` is imported.
- 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 `:_ :` S_i `->` `Attribute` for each attribute a_i.
- For each subclass relation C `<` C' a subsort declaration

  `subsort` C `<` C' `.`

  is introduced, and the set of attributes for objects of class C is completed with those of C'.
- 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 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 `<` O `:` C `|` $\ldots$ `>` appearing in a rule is translated into `<` O `:` X `|` $\ldots$`,` $Atts$ `>` 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 the $\overrightarrow{a}$ 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

$$\ldots \langle O : C \mid \overrightarrow{al : vl}, \overrightarrow{ab : vb} \rangle \ldots \longrightarrow \ldots \langle O : C \mid \overrightarrow{ab : vb'}, \overrightarrow{ar : vr'} \rangle \ldots$$

[3] We have simplified the transformation of object-oriented modules into system modules that originally appeared in [46].

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

$$\ldots \langle O : X \mid \overrightarrow{al : vl}, \overrightarrow{ab : vb}, \overrightarrow{ar : x}, \overrightarrow{ac : x'}, Atts \rangle \ldots$$
$$\longrightarrow \ldots \langle O : X \mid \overrightarrow{al : vl}, \overrightarrow{ab : vb'}, \overrightarrow{ar : vr}, \overrightarrow{ac : x'}, Atts \rangle \ldots$$

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 `ACCNT` module are interpreted here—according to the conventions already explained—in a form that can be inherited by subclasses of `Accnt` that could be defined later. Thus, `SavAccnt` inherits the rewrite rules for crediting and debiting accounts, and for transferring funds between accounts that had been defined for `Accnt`.

Let us illustrate the treatment of class inheritance with the system module resulting from the transformation of the module `SAV-ACCNT` introduced previously.

```
mod SAV-ACCNT is
  including CONFIGURATION+ .
  including ACCNT .
  sorts SavAccnt .
  subsort SavAccnt < Cid .
  subsort SavAccnt < Accnt .
  op SavAccnt : -> SavAccnt .
  op rate :_ : Int -> Attribute .
endm
```

Note that by translating a rule like `credit` above

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

into its corresponding desugared form

```
rl [credit] :
  credit(A, M)
  < A : V0@:Accnt | bal : N, V1@:AttributeSet >
  => < A : V0@:Accnt | bal : (N + M), V1@:AttributeSet > .
```

it is guaranteed that the rule will be applicable to objects of class `Accnt` as well as of any of its subclasses.

Note also that a rule like `change-age` (discussed in Section 14.1.4)

```
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`, but also guarantee that it is only applied to well-formed objects, that is, to objects with all the required attributes.

Part III

Reference

Chapter 15

Complete List of Maude Commands

We use curly bracket pairs, '{' and '}', to enclose optional syntax.

15.1 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.

-no-wrap Disables the automatic line wrapping of output.

15.2 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. Examples:

```
reduce 6 * 7 == 42 .
reduce in QID : conc('a, 'b) .
```

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 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**. Examples:

```
rewrite 6 * 7 == 42 .
rewrite in FOO : f(6, g(a, b)) .
rewrite [50] f(6, g(a, b)) .
rewrite [1] in BAR : h(a) .
```

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 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. If the bound is omitted, infinity is assumed. **frewrite** may be abbreviated to **frew**. Examples:

```
frewrite 6 * 7 == 42 .
frewrite in FOO : f(6, g(a, b)) .
frewrite [50, 2] f(6, g(a, b)) .
frewrite [100, 4] in BAR : h(a) .
```

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, and 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 haven't 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:

(a) If position-fair rewriting stops mid traversal, then the sort of the (incompletely reduced) result has not been calculated and is printed as `(sort not calculated)`.

(b) Position-fair rewriting is not substitution fair; this is particularly apparent if you have a multiset of messages and objects.

erewrite {[***bound*** {,***number***}]} {**in** ***module*** **:**} ***term*** **.** Works like `frewrite` 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 that 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 so that

```
frewrite [n, k] t .
continue m .
```

produces the same final answer as

```
frewrite [s, k] t .
```

when `s = n + m`.

loop {**in** ***module*** **:**} ***term*** **.** This command is used to initialize the read-eval-print loop in a module importing `LOOP-MODE` (see Section 11.1). 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 on 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**** **)** This command is used to input a list of identifiers to the loop in a module importing `LOOP-MODE` (see Section 11.1). If the current module has not changed since the last rewriting command, the result of previous rewrites has a loop constructor symbol on 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 on 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.

15.3 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*** { **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. If the upper bound clause is omitted, infinity is assumed. Example:

```
match [5] in FOO : +(X, *(X, Y)) <=? +(*(a, b), *(c, d)) .
```

xmatch {[***number***]} {**in** ***module*** **:**} ***pattern*** `<=?` ***subject*** { **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 (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. If only part of the subject was matched, that part is given. Example:

```
xmatch +(*(P, Q), *(X, Y)) <=? +(*(a, b), *(c, d), *(a, e)) .
```

15.4 Searching commands

search {[***bound***]} {**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*. Possible values for *searchtype* are

`=>1`	one step proof
`=>+`	one or more steps proof
`=>*`	zero or more steps proof
`=>!`	only canonical final states allowed

The maximum number of solutions found is given by *bound* which defaults to infinity. Examples:

```
search [2] in FOO : a * d + b * d + c * d + a * e + =>+ X * Y .
search a * d + b * d + c * d + a * e + =>* X * Y such that X == Y + 2 .
```

show search graph . Displays the search graph generated by the last search.

show path *number* . Displays the path to a given state 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.

15.5 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.

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 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 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.

15.6 Print option commands

set print mixfix on . / set print mixfix off . Controls whether operators with mixfix syntax are printed in 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.

15.7 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 15.2. 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 15.2. 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.

15.8 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 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).

15.9 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.

15.10 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 .
```

debug reduce {**in** ***module*** **:**} ***term*** **.** Works just like the **reduce** command in Section 15.2, except that it drops into the debugger before executing the first rewrite.

debug rewrite {[***number***]} {**in** ***module*** **:**} ***term*** **.** Works just like the **rewrite** command in Section 15.2, except that it drops into the debugger before executing the first rewrite.

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.

15.11 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.

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.

15.12 System 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
```

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
```

load *file-name* Performs the same job as `in` but does not produce detailed output as modules are entered. Example:

```
load ../Examples/foo.maude
```

eof Causes the interpreter to respond as if it had reached the end of file.

quit Causes the interpreter to exit.

Chapter 16

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 "`x`" denotes `x` literally. As an application of this notation, `A( , A)*` indicates a nonempty list of `A`'s separated by commas. Finally, `%%%` indicates comments in the syntactic description, as opposed to comments in the Maude code.

16.1 The grammar

```
⟨MaudeTop⟩ ::=
    ( ⟨SystemCommand⟩ | ⟨Command⟩ | ⟨DebuggerCommand⟩ |
      ⟨Module⟩ | ⟨Theory⟩ | ⟨View⟩ )+

⟨SystemCommand⟩ ::= in ⟨FileName⟩ | load ⟨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 [ [ ⟨Nat⟩ [ , ⟨Nat⟩ ] ] ] [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    [ debug ] erewrite [ [ ⟨Nat⟩ [ , ⟨Nat⟩ ] ] ] [ in ⟨ModId⟩ : ] ⟨Term⟩ . |
    ( match | xmatch ) [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ] ⟨Term⟩ <=? ⟨Term⟩
        [ such that ⟨Condition'⟩ ] . |
    search [ [ ⟨Nat⟩ ] ] [ in ⟨ModId⟩ : ] ⟨Term⟩ ⟨SearchType⟩ ⟨Term⟩
        [ such that ⟨Condition'⟩ ] . |
    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 . |
    set ⟨SetOption⟩ ( on | off ) .

⟨ShowItem⟩ ::= module | all | sorts | ops | vars | mbs |
    eqs | rls | 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

⟨TraceOption⟩ ::= condition | whole | substitution | select |
    mbs | eqs | rls | rewrite | body

⟨DebuggerCommand⟩ ::= resume . | abort . | step . | where .

⟨Module⟩ ::= fmod ⟨ModId⟩ [ ⟨ParameterList⟩ ] is ⟨ModElt⟩* endfm |
    mod ⟨ModId⟩ [ ⟨ParameterList⟩ ] is ⟨ModElt'⟩* endfm

⟨Theory⟩ ::= fth ⟨ModId⟩ is ⟨ModElt⟩* endfth |
    th ⟨ModId⟩ is ⟨ModElt'⟩* endth

⟨View⟩ ::= view ⟨ViewId⟩ 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⟩ .

⟨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⟩

⟨ToPartRenamingItem⟩ ::= to ⟨OpForm⟩ [ ⟨Attr⟩ ]

⟨Arrow⟩ ::= -> | ~>

⟨Type⟩ ::= ⟨Sort⟩ | ⟨Kind⟩

⟨Kind⟩ ::= [ ⟨Sort⟩ ( , ⟨Sort⟩ )* ]

⟨Sort⟩ ::= ⟨SortId⟩ | ⟨Sort⟩ { ⟨Sort⟩ ( , ⟨Sort⟩ )* }

⟨ModElt'⟩ ::= ⟨ModElt⟩ |
    ⟨Statement'⟩ [ ⟨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'⟩

⟨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 |
        metadata ⟨StringId⟩
        strat ( ⟨Nat⟩+ ) |
        poly ( ⟨Nat⟩+ ) |
        frozen [ ( ⟨Nat⟩+ ) ] |
        prec ⟨Nat⟩ |
        gather ( ( e | E | & )+ ) |
        format ( ⟨Token⟩+ ) |
        special ( ⟨Hook⟩+ ) )+ ]

⟨StatementAttr⟩ ::=
    [ ( nonexec | otherwise |
        metadata ⟨StringId⟩
        label ⟨LabelId⟩ )+ ]

⟨Hook⟩ ::= id-hook ⟨Token⟩ [ ( ⟨TokenString⟩ ) ] |
    ( op-hook | term-hook ) ( ⟨TokenString⟩ )

⟨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
⟨OpId⟩        %%% identifier possibly with underscores
⟨OpForm⟩ ::= ⟨OpId⟩ | ( ⟨OpForm⟩ ) | ⟨OpForm⟩+
⟨Nat⟩         %%% a natural number
⟨Term⟩ ::= ⟨Token⟩ | ( ⟨Term⟩ ) | ⟨Term⟩+
⟨Token⟩       %%% Any symbol other than ( or )
⟨TokenString⟩ ::= ⟨Token⟩ | ( ⟨TokenString⟩ ) | ⟨TokenString⟩*
⟨LabelId⟩     %%% simple identifier
```

In parsing module expressions, instantiation has higher precedence than renaming, which in turn has higher precedence than summation.

16.2 Synonyms

```
sort = sorts
subsort = subsorts
var = vars
```

Command only synonyms:

```
advise = advisory = advisories
alias = aliases
cmd = command
cond = condition
cont = continue
eqs = eq
erew = erewrite
flat = flattened
frew = frewrite
kinds = components
label = labels
mbs = mb
paren = parens = parentheses
q = quit
rat = rational
red = reduce
rew = rewrite
rls  = rl = rule = rules
s.t. = such that
subst = substitution
```

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
```

16.3 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 [39, Section A2.5.2].

Bibliography

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] David Basin, Manuel Clavel, and José Meseguer. Reflective metalogical frameworks. *ACM Transactions on Computational Logic*, 5(3):528–576, 2004.

[3] 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.

[4] Hans-Juergen Boehm, Russell R. Atkinson, and Michael F. Plass. Ropes: An alternative to strings. *Software Practice and Experience*, 25(12):1315–1330, 1995.

[5] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.

[6] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[7] Christiano O. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Departamento de Informática, Pontificia Universidade Católica de Rio de Janeiro, Brasil, 2001.

[8] 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.

[9] 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.

[10] Feng Chen, Grigore Roşu, and Ram Prasad Venkatesan. Rule-based analysis of dimensional safety. 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 197–207. Springer, 2003.

[11] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2001.

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

[13] 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.uiuc.edu/maude1/tutorial/`.

[14] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Towards Maude 2.0. In 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*, pages 294–315. Elsevier, 2000. `http://www.sciencedirect.com/science/journal/15710661`.

[15] 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.

[16] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998. `http://maude.cs.uiuc.edu/papers/`.

[17] 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.

[18] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In 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*, pages 126–148. Elsevier, 1996. `http://www.sciencedirect.com/science/journal/15710661`.

[19] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

[20] Manuel Clavel, José Meseguer, and Miguel Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. In Fabio Gadducci and Ugo Montanari, editors, *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*, pages 91–107. Elsevier, 2002. `http://www.sciencedirect.com/science/journal/15710661`.

[21] Evelyn Contejean and Hervé Devie. An efficient incremental algorithm for solving systems of linear diophantine equations. *Information and Computation*, 113(1):143–172, 1994.

[22] 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.

[23] Francisco Durán. Termination checker and Knuth-Bendix completion tools for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, 2000. `http://maude.cs.uiuc.edu/papers/`.

[24] Francisco Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, Spain, 1999. `http://maude.cs.uiuc.edu/papers/`.

[25] 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.

[26] Francisco Durán and José Meseguer. A Church-Rosser checker tool for Maude equational specifications. Manuscript,Computer Science Laboratory, SRI International, 2000. `http://maude.cs.uiuc.edu/papers/`.

[27] Francisco Durán and José Meseguer. An extensible module algebra for Maude. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 174–195. Elsevier, 1998. `http://www.sciencedirect.com/science/journal/15710661`.

[28] Francisco Durán and José Meseguer. The Maude specification of Full Maude. Technical report, Computer Science Laboratory, SRI International, February 1999. `http://maude.cs.uiuc.edu/papers/`.

[29] Francisco Durán and José Meseguer. Parameterized theories and views in Full Maude 2.0. In 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*, pages 316–338. Elsevier, 2000. `http://www.sciencedirect.com/science/journal/15710661`.

[30] Francisco Durán and José Meseguer. Structured theories and institutions. *Theoretical Computer Science*, 309:357–380, 2003.

[31] Steven Eker. Fast matching in combination of regular equational theories. In 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*, pages 90–109. Elsevier, 1996. `http://www.sciencedirect.com/science/journal/15710661`.

[32] Steven Eker. Term rewriting with operator evaluation strategies. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 311–330. Elsevier, 1998. `http://www.sciencedirect.com/science/journal/15710661`.

[33] 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.

[34] Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, José Meseguer, and Kemal Sonmez. Pathway logic: Symbolic analysis of biological signaling. In Russ B. Altman, A. Keith Dunker, Lawrence Hunter, and Teri E. Klein, editors, *Proceedings of the 7th Pacific Symposium on Biocomputing (PSB 2002), Lihue, Hawaii, USA, January 3-7, 2002*, pages 400–412, January 2002. `http://helix-web.stanford.edu/psb02/`.

[35] Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, and Carolyn Talcott. Pathway Logic: Executable models of biological networks. In 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*, pages 125–142. Elsevier, 2002. `http://www.sciencedirect.com/science/journal/15710661`.

[36] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In 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*, pages 143–168. Elsevier, 2002. `http://www.sciencedirect.com/science/journal/15710661`.

[37] Kokichi Futatsugi and Razvan Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.

[38] 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.

[39] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[40] Nirman Kumar, Koushik Sen, José Meseguer, and Gul Agha. Probabilistic rewrite theories: unifying models, logics, and tools. Manuscript, University of Illinois at Urbana-Champaign, March 2003.

[41] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer-Verlag, 1992.

[42] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd. Edition*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Technical Report SRI-CSL-93-05, August 1993.

[43] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[44] Ian A. Mason and Carolyn L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 228(1), 1999.

[45] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[46] 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. MIT Press, 1993.

[47] 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.

[48] José Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In Scott F. Smith and Carolyn L. Talcott, editors, *Proceedings IFIP Conference on Formal Methods for Open Object-Based Distributed Systems IV, FMOODS 2000, September 6–8, 2000, Stanford, California, USA*, pages 89–117. Kluwer Academic Publishers, 2000.

[49] José Meseguer and Grigore Roşu. Towards behavioral Maude: Behavioral membership equational logic. In Lawrence S. Moss, editor, *CMCS'2002, Coalgebraic Methods in Computer Science, Grenoble, France, 6-7 April 2002*, volume 65(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. `http://www.sciencedirect.com/science/journal/15710661`.

[50] Peter Csaba Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000. `http://maude.cs.uiuc.edu/papers/`.

[51] Peter Csaba Ölveczky and José Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In 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*, pages 361–382. Elsevier, 2000. `http://www.sciencedirect.com/science/journal/15710661`.

[52] Peter Csaba Ölveczky and José Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[53] 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.

[54] J. F. Quesada. *The SCP parsing algorithm based on syntactic constraint propagation*. PhD thesis, University of Seville, 1997.

[55] J. F. Quesada. The Maude parser: Parsing and meta-parsing β-extended context-free grammars. Technical Report, SRI International, Computer Science Laboratory, 1999.

[56] Grigore Rosu, Ram Prasad Venkatesan, Jon Whittle, and Laurentiu Leustean. Certifying optimality of state estimation programs. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 301–314. Springer, 2003.

[57] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[58] Andrzej Tarlecki, editor. *Solving systems of linear diophantine equations: An algebraic approach*, volume 520 of *Lecture Notes in Computer Science*. Springer, 1991.

[59] Ana Paula Tomás. *On Solving Linear Diophantine Constraints*. PhD thesis, Universidade do Porto, 1997.

[60] Arie van Deursen. *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.

[61] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996.

[62] Alberto Verdejo. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.

[63] Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in Maude 2. In 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*, pages 263–281. Elsevier, 2002. `http://www.sciencedirect.com/science/journal/15710661`.

[64] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.