A tutorial on specifying data structures in Maude[⋆]

Narciso Martí-Oliet, Miguel Palomino, and Alberto Verdejo

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
narciso,miguelpt,alberto@sip.ucm.es

Abstract. This tutorial describes the equational specification of a series of typical data structures in Maude. We start with the well-known stacks, queues, and lists, to continue with binary and search trees. Not only are the simple versions considered but also advanced ones such as AVL and 2-3-4 trees. The operator attributes available in Maude allow the specification of data based on constructors that satisfy some equational properties, like concatenation of lists which is associative and has the empty list as identity, as opposed to the free constructors available in other functional programming languages. Moreover, the expressive version of equational logic in which Maude is based, namely membership equational logic, allows the faithful specification of types whose data are defined not only by means of constructors, but also by the satisfaction of additional properties, like sorted lists or search trees. In the second part of the paper we describe the use of an inductive theorem prover, ITP, which itself is developed and integrated in Maude by means of the powerful metalevel and metalanguage features offered by the latter, to prove properties of the data structures. This is work in progress because the ITP is still under development and, as soon as the data gets a bit complex, the proof of their properties gets even more complex.

Keywords: Data structures, algebraic specification, membership equational logic, Maude, inductive theorem proving.

1 Introduction

Maude is a declarative language and system based on rewriting logic [6, 5]. Even though both the language and the system keep being improved, they reached maturity with the public release of version 2 in the summer of 2003. Since then, Maude is being used throughout the world in teaching and research, being specially useful for the specification and prototyping of logical systems, programming languages, and computational systems in general. However, there is still a lack of common libraries that could be shared and reused.

This tutorial tries to contribute to fill this gap by providing a library of typical data structures specified in Maude. This is accomplished by updating

⋆ Research partially supported by MCyT Spanish projects *MIDAS* (TIC2003–01000) and *MELODIAS* (TIC2002–01167).

and considerably extending the set of specifications that were available in the tutorial distributed with version 1 of Maude [4]. More specifically, we start by describing well-known versions of basic datatypes such as stacks, queues, and lists; we then continue with several versions of trees, including binary, general, and search trees; we do not consider only the simple versions, but also advanced ones such as AVL, 2-3-4, and red-black trees; finally, we describe an abstract version of priority queues and a more concrete one based on leftist trees.

For all of these specifications we do not need to consider rewriting logic in its full generality, but just its equational sublogic, namely, membership equational logic [11] (from the Maude language point of view, all of our specifications are *functional modules*). A very important point is that the expressivity of this equational logic allows the faithful specification of types whose data are defined not only by means of constructors, but also by the satisfaction of additional properties, like sorted lists, search trees, balanced trees, etc. We will see along the paper how this is accomplished by means of membership assertions that equationally characterize the properties satisfied by the corresponding data.

All the datatypes that we consider are generic, that is, they are constructions on top of other datatypes that appear as parameters in the construction. Therefore, our specifications are *parameterized* and, for this reason, we use Full Maude [6], which provides powerful mechanisms for parameterization based on *theories* that describe the requirements that a data type must satisfy for the construction to make sense. For example, lists can be constructed on top of any data whatsoever, but sorted lists only make sense for data that have a total order; for a binary operation to be a total order, several properties have to be satisfied, which are written in the corresponding parameter theory as equations.

We assume some knowledge about the data structures that are specified. There are many textbooks that describe well-known imperative and object-oriented implementations [9, 2, 16]. Less known, but very useful for our purposes, are implementations in functional programming languages such as ML or Haskell [12, 15, 14]; in some cases, our equations are very similar to the ones given in such texts. On the other hand, we do not assume much knowledge about Maude and thus we describe the main features as they come out along with the specifications.

As mentioned before, here we do not consider at all rule-based programming in Maude. For an introduction to those features, we refer the interested reader to the paper [13].

All the code in this paper and more can be found in the web page [10].

2 Review of main features

2.1 Functional modules

A functional module in Maude corresponds to an equational theory in membership equational logic.

Both the logic and the language are typed, and *types* are declared by means of the keywords `sort` or `sorts`. Then each operator, introduced by means of the keyword `op`, has to be declared together with the sorts of its arguments and the sort of its result. There is an inclusion relation between types, which is described by means of `subsort` declarations. Operators can be *overloaded.*

With typed variables (that can either be declared separately, or used on-the-fly annotated with the corresponding sort) and operators, we can build terms in the usual way. A given term can have many different sorts, because of the subsorting and overloading. Under some easy-to-satisfy requirements, a term has a least sort. Terms are used to form

- *membership assertions* $t : s$ (introduced with keyword `mb`), stating that the term t has sort s, and
- *equations* $t = t'$ (introduced with keyword `eq`), stating that the meaning of terms t and t' is the same.

Both memberships and equations can be conditional, with respective keywords `cmb` and `ceq`. Conditions are formed by a conjunction (written `/\`) of equations and memberships.

Computation in a functional module is done by using the equations as simplification rules from left to right until a canonical form is found. For this to be meaningful, the variables in the righthand side of an equation have to be included among those in the lefthand side (a generalization is provided by means of matching equations in conditions, as we will see later); moreover, the set of equations must be terminating and confluent. This guarantees that all terms will simplify to a unique canonical form [1].

Some equations, like commutativity, are not terminating, but nonetheless they are supported by means of *operator attributes*, so that Maude performs simplification modulo the equational theories provided by such attributes, that can be associativity, commutativity, identity, and idempotence. Properties such as termination and confluence shoud be understood in this more general context of simplification modulo some equational theories.

Modules can be imported in different modes. The most important one is `protecting` that asserts that all the information in the imported module does not change because of the importation; more specifically, different data in the imported module are not identified in the importing module, and no new data are added to the imported sorts. When this is not case, the importation mode can be `including`.

2.2 Parameterization

As we have already mentioned, parameterized datatypes use *theories* to specify the requirements that the parameter must satisfy. A (functional) theory is also a membership equational specification but since its equations are not used for equational simplication, they need not satisfy any requirement about variables in the righthand side, confluence, or termination.

The simplest theory is the one requiring just the existence of a sort, as follows:

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

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

A more complex theory is the following, requiring a (strict) total order over elements of a given sort. Notice the *new* variable `E2` in the righthand side of the first conditional equation. This makes this equation non-executable, as stated by the attribute `nonexec` next to the equation.

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

This theory imports in `protecting` mode the predefined module `BOOL` of Boolean values, meaning that the Boolean values are not disturbed in any way.

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

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

where `X :: TRIV` denotes that `X` is the label of the formal parameter, and that it must be instantiated with modules satisfying the requirement expressed by the theory `TRIV`. The way to express this instantiation is by means of *views*. A view shows how a particular module satisfies a theory, by mapping sorts and operations in the theory to sorts and operations (or, more generally, terms) in the target module, in such a way that the induced translations on equations and membership axioms are provable in the module. In general, this requires theorem proving that is not done by the system, but is instead delegated to the ITP tool (see Section 4). However, in many simple cases the proof of obligations associated to views are completely obvious, as for example in the following view from the theory `TRIV` to the predefined module `NAT` of natural numbers, where, since `TRIV` has no equations, no proof obligations are generated.

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

Then, the module expression `LIST(Nat)` denotes the instantiation of the parameterized module `LIST(X :: TRIV)` by means of the above view `Nat`. Views can also go from theories to theories, as we will see later in Section 3.4. For more information on parameterization and how it is implemented in the Maude system, the reader is referred to [7, 6].

3 Data Structures

3.1 Stacks

We begin our series of datatype specifications with stacks. Since stacks can be built over any datatype, the requirement theory is `TRIV`. The main sort is `Stack(X)`; notice that its name makes explicit the label of the parameter. In this way, when the module is instantiated with a view, like for example `Nat` above (from `TRIV` to `NAT`), the sort name is also instantiated becoming `Stack(Nat)`, which makes clear that the data are stacks of natural numbers.

The sorts and operations of the theory are used in the body of the parameterized module, but sorts are qualified with the label of the formal parameter; thus in this case the parameter sort `Elt` becomes `X@Elt` in the `STACK` parameterized module. In this way, although not needed in our examples, one can have several different parameters satisfying the same parameter theory.

The only subtle point in a stack specification is that the `top` operation is partial, because it is not defined on the empty stack. In our specification, we use a subsort `NeStack(X)` of non-empty stacks to handle this situation. Then, `push` becomes a *constructor* of non-empty stacks, while both `empty` and `push` (the latter via subsorting) are *constructors* of stacks; notice the `ctor` attribute of those operators, indicating that they are constructors. Now, the `top` and `pop` operations are defined as total with domain `NeStack(X)`.

Finally, all modules import implicitly the predefined `BOOL` module, and therefore we can use the sort `Bool` and the Boolean values `true` and `false` when necessary.

```
(fmod STACK(X :: TRIV) is
   sorts NeStack(X) Stack(X) .
   subsort NeStack(X) < Stack(X) .
   op empty : -> Stack(X) [ctor] .
   op push : X@Elt Stack(X) -> NeStack(X) [ctor] .
   op pop : NeStack(X) -> Stack(X) .
   op top : NeStack(X) -> X@Elt .
   op isEmpty : Stack(X) -> Bool .
   var S : Stack(X) .   var E : X@Elt .
   eq pop(push(E,S)) = S .
   eq top(push(E,S)) = E .
   eq isEmpty(empty) = true .
   eq isEmpty(push(E,S)) = false .
endfm)
```

The following view maps the theory `TRIV` to the predefined module `INT` of integers, and is then used in an example of term reduction, invoked with the Maude command `red`.

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

```
Maude> (red in STACK(Int) : top(push(4,push(5,empty))) .)
result NzNat : 4
```

Notice that Maude computes the least sort of the result. In the following sections, for lack of space, we do not show reduction examples, but they can be found in [10].

3.2 Queues

The specification for queues is very similar to the one for stacks and therefore it is not included here. It is available in [10].

3.3 Lists

We are going to specify lists in two ways, by using different sets of constructors in each case. In both cases, lists are parameterized with respect to the `TRIV` theory.

The first version uses the two standard free constructors that can be found in many functional programming languages: the empty list *nil*, here denoted `[]`, and the *cons* operation that adds an element to the beginning of a list, here denoted with the mixfix syntax `_:_`. As usual, `head` and `tail` are the selectors associated to this constructor. Since they are not defined on the empty list, we avoid their partiality in the same way as we have done for stacks (see Section 3.1) by means of a subsort `NeList` of non-empty lists. The remaining operations on lists (defined as usual by structural induction on the two constructors) concatenate two lists, calculate the length of a list, and reverse a list; the second one of those has a result of sort `Nat` that comes from the imported (in `protecting` mode) predefined module `NAT`.

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

```
(fmod LIST(X :: TRIV) is
   protecting NAT .
   sorts NeList(X) List(X) .
   subsort NeList(X) < List(X) .
   op `[`] : -> List(X) [ctor] .
   op _:_ : X@Elt List(X) -> NeList(X) [ctor] .
   op tail : NeList(X) -> List(X) .
   op head : NeList(X) -> X@Elt .
   op _++_ : List(X) List(X) -> List(X) .
   op length : List(X) -> Nat .
   op rev : List(X) -> List(X) .
   var E : X@Elt .   var N : Nat .   vars L L' : List(X) .
   eq tail(E : L) = L .
   eq head(E : L) = E .
   eq [] ++ L = L .
   eq (E : L) ++ L' = E : (L ++ L') .
   eq length([]) = 0 .
   eq length(E : L) = 1 + length(L) .
   eq rev([]) = [] .
   eq rev(E : L) = rev(L) ++ (E : []) .
endfm)
```

Another way to generate lists is to begin with the empty list and the singleton lists, and then use the concatenation operation to get bigger lists. However, concatenation cannot be a free list constructor, because it satisfies an associativity equation. This equation is not declared directly as such, but as an operator attribute `assoc`. Moreover, there is also an identity relationship of concatenation with respect to the empty list, which is expressed by means of the attribute `id: []`. It is very convenient here to use empty juxtaposition syntax (declared as `__` in the following module) for the concatenation operator as constructor; in this way, the list of integers `1 : 2 : 3 : []` in the previous notation now becomes simply `1 2 3`.

Notice how the singleton lists are identified with the corresponding elements (by means of a subsort declaration `X@Elt < NeList(X)`) and also how the concatenation operator is subsort overloaded, having one declaration for non-empty lists and another one for lists, both with the same attributes. There are two more possibilities of concatenation overloading (`NeList List -> NeList` and `List NeList -> NeList`) but they are unnecessary in this case because of the attribute for identity. Finally, notice that operator attributes in overloaded operations have to coincide, even though, by reading alone the second declaration for concatenation, it may sound a bit strange to say that the empty list is an identity for an operation only defined on non-empty lists. When there are many overloaded declarations for an operator, it is possible to use the operator attribute `ditto` to implicitly repeat the attributes without having to write all of them explicitly again.

Because of the operator attributes, congruence classes over which simplification takes place are computed modulo associativity and identity. Therefore, we only need two equations to completely specify the behavior of defined operations like `length` and `rev`; the singleton case is included in the `E L` case by instantiating the variable `L` with the constant `nil` and applying the equational attribute for identity. Notice that in this way, even though we have changed the set of list constructors, we are not changing so much (except for the notation) the style of the definitions by structural induction of the remaining operations; the case `E L` corresponds to the *cons* case in the previous specification.

```
(fmod LIST-CONCAT(X :: TRIV) is
   protecting NAT .
   sorts NeList(X) List(X) .
   subsorts X@Elt < NeList(X) < List(X) .
   op `[`] : -> List(X) [ctor] .
   op __ : List(X) List(X) -> List(X) [ctor assoc id: `[`]] .
   op __ : NeList(X) NeList(X) -> NeList(X) [ctor assoc id: `[`]] .
   op tail : NeList(X) -> List(X) .
   op head : NeList(X) -> X@Elt .
   op length : List(X) -> Nat .
   op rev : List(X) -> List(X) .
   var E : X@Elt .   var L : List(X) .
   eq tail(E L) = L .
   eq head(E L) = E .
   eq length(nil) = 0 .
   eq length(E L) = 1 + length(L) .
   eq rev(nil) = nil .
   eq rev(E L) = rev(L) E .
endfm)
```

3.4 Ordered lists

In order-sorted equational specifications, subsorts must be defined by means of constructors, but it is not possible to have a subsort of ordered lists, for example, defined by a property over lists; a more expressive formalism is needed. *Membership equational logic* allows subsort definition by means of conditions involving equations and/or sort predicates. In this example we use this technique to define a subsort `OrdList`, containing *ordered lists*,[1] of the sort `List` of lists, which is imported from the module `LIST` in Section 3.3. Notice the three (conditional) membership axioms defining the sort `OrdList`: the empty and singleton lists are always ordered, and a longer list is ordered when the first element is less than or equal to the second, and the list without the first element is also ordered.

Parameterized ordered lists need a stronger requirement than `TRIV`, because we need a total order over the elements to be ordered. The theory `TOSET<` that we saw in Section 2.2 requires a strict total order on the elements; since repetitions do not give any trouble for sorting a list, we can have a `_<=_` operation in the requirement. We can also import theories (in `including` mode), and thus we can define our theory as follows:

```
(fth TOSET<= is
  including TOSET< .
  op _<=_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X <= Y = X < Y or X == Y .
endfth)
```

The parameterized module for ordered lists is going to import the parameterized list module. However, note that we want lists for a totally ordered set, instead of lists over any set; therefore, we *partially instantiate* `LIST` with a view from the theory `TRIV` to the theory `TOSET<=`

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

[1] We prefer "ordered list" over "sorted list" because "sort" is already used to refer to types in this context.

and we are still left with a parameterized module and corresponding dependent sorts, but now with respect to the `TOSET<=` requirement. This is the reason justifying the notation `LIST(Toset)(X)` in the `protecting` importation below, as well as `NeList(Toset)(X)` and `List(Toset)(X)` in the names of the imported sorts.

As part of this module for ordered lists, we also define several well-known sorting operations: `insertion-sort`, `quicksort`, and `mergesort` (the following code only includes the first one, while the other two can be found in the more complete version available in [10]). Each of them uses appropriate auxiliary operations whose behavior is the expected one; for example, `insertion-sort` recursively sorts the list without the first element, and then calls `insert-list`, which inserts the missing element in the correct position.

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

```
(fmod ORD-LIST(X :: TOSET<=) is
   protecting LIST(Toset)(X) .
   sorts OrdList(X) NeOrdList(X) .
   subsorts NeOrdList(X) < OrdList(X) NeList(Toset)(X) < List(Toset)(X) .
   op insertion-sort : List(Toset)(X) -> OrdList(X) .
   op insert-list : OrdList(X) X@Elt -> OrdList(X) .
   vars N M : X@Elt .            vars L L' : List(Toset)(X) .
   vars OL OL' : OrdList(X) .    var NEOL : NeOrdList(X) .
   mb [] : OrdList(X) .
   mb (N : []) : NeOrdList(X) .
   cmb (N : NEOL) : NeOrdList(X) if N <= head(NEOL) .
   eq insertion-sort([]) = [] .
   eq insertion-sort(N : L) = insert-list(insertion-sort(L), N) .
   eq insert-list([], M) = M : [] .
   ceq insert-list(N : OL, M) = M : N : OL if M <= N .
   ceq insert-list(N : OL, M) = N : insert-list(OL, M) if M > N .
 endfm)
```

3.5 Multisets

In the same way as associativity and identity for concatenation provide structural axioms for lists (or strings), we can specify multisets by considering a union constructor (written again with empty juxtaposition syntax) that satisfies associativity, commutativity (because now order between elements does not matter), and identity structural axioms, all declared as attributes. Singleton multisets are identified with elements, as we also did for lists, by declaring `X@Elt` as a subsort of `Mset(X)`.

```
(fmod MULTISET(X :: TRIV) is
   protecting NAT .
   sort Mset(X) .
   subsort X@Elt < Mset(X) .
   op empty : -> Mset(X) [ctor] .
   op __ : Mset(X) Mset(X) -> Mset(X) [ctor assoc comm id: empty] .
   op size : Mset(X) -> Nat .
   op mult : X@Elt Mset(X) -> Nat .
   op delete : X@Elt Mset(X) -> Mset(X) .
   op is-in : X@Elt Mset(X) -> Bool .
   vars E E' : X@Elt .   var S : Mset(X) .
   eq size(empty) = 0 .
   eq size(E S) = 1 + size(S) .
   eq mult(E, empty) = 0 .
   eq mult(E, E S) = 1 + mult(E, S) .
   ceq mult(E, E' S) = mult(E, S) if E =/= E' .
   eq delete(E, empty) = empty .
   eq delete(E, E S) = delete(E, S) .
   ceq delete(E, E' S) = E' delete(E, S) if E =/= E' .
   eq is-in(E, S) = mult(E, S) > 0 .
 endfm)
```

3.6 Sets

Analogously to obtaining multisets from lists by adding commutativity, one can get sets from multisets by adding idempotence. In the current version of Maude, the operator attributes for associativity and idempotence are not compatible, but this is easily solved by adding an operator attribute for associativity and an explicit equation for idempotence. Since the specification of sets is very similar to the previous one for multisets, we refer to [10].

3.7 Binary trees

Binary trees (parameterized with respect to `TRIV`) are built with two free constructors: the empty tree, denoted `empty`, and an operation that puts an element as root above two given trees, its left and right children, denoted `_[_]_`. The three selectors associated to this constructor (`root`,`left`, and `right`) only make sense for non-empty trees, that belong to the corresponding subsort.

The operation that calculates the depth (or height) of a binary tree calls a `max` operation on natural numbers in the recursive non-empty case to obtain the maximum of two such numbers. Since this operation is not provided in the predefined module `NAT`, we import a module `NAT-MAX` that has previously added the `max` operation to `NAT`.

Finally, we also have three operations that calculate the standard binary tree traversals, with `List(X)` as value sort (this is the reason this module imports the

LIST module). Since all of them have the same rank, they are declared together by means of the keyword ops.

```
(fmod BIN-TREE(X :: TRIV) is
   protecting LIST(X) .   protecting NAT-MAX .
   sorts NeBinTree(X) BinTree(X) .
   subsort NeBinTree(X) < BinTree(X) .
   op empty : -> BinTree(X) [ctor] .
   op _`[_`]_ : BinTree(X) X@Elt BinTree(X) -> NeBinTree(X) [ctor] .
   ops left right : NeBinTree(X) -> BinTree(X) .
   op root : NeBinTree(X) -> X@Elt .
   op depth : BinTree(X) -> Nat .
   ops preorder inorder postorder : BinTree(X) -> List(X) .
   var E : X@Elt .   vars L R : BinTree(X) .   vars NEL NER : NeBinTree(X) .
   eq left(L [E] R) = L .
   eq right(L [E] R) = R .
   eq root(L [E] R) = E .
   eq depth(empty) = 0 .
   eq depth(L [E] R) = 1 + max(depth(L), depth(R)) .
   eq preorder(empty) = [] .
   eq preorder(L [E] R) = E : (preorder(L) ++ preorder(R)) .
   eq inorder(empty) = [] .
   eq inorder(L [E] R) = inorder(L) ++ (E : inorder(R)) .
   eq postorder(empty) = [] .
   eq postorder(L [E] R) = postorder(L) ++ (postorder(R) ++ (E : [])) .
 endfm)
```

3.8 General trees

General trees can have a variable number of children for each node. One can specify them by using an auxiliary datatype of *forests* that behave like lists of trees. Since otherwise the parameterization and specification techniques do not differ from the ones we have already described before, we do not include the specification here, and instead refer again to [10].

3.9 Binary search trees

This example is similar in philosophy to the one for ordered lists, but is more complex. We specify a subsort of *(binary) search trees* by using several (conditional) membership axioms over terms of the sort BinTree of binary trees defined in Section 3.7.

Although we allowed repeated elements in an ordered list, this should not be the case in a search tree, where all nodes must contain different values. A binary search tree is either the empty binary tree or a non-empty binary tree such that all elements in the left child are smaller than the element in the root, all elements in the right child are bigger than it, and both the left and right children are also binary search trees. This is checked by means of auxiliary operations that calculate the minimum and maximum element in a non-empty search tree, and that are also useful when deleting an element. Again, the most important point is that membership equational logic allows us both to define the corresponding subsort by means of membership assertions (we consider five cases in the specification below) and to assign typings in the best possible way to all the operations defined for this datatype.

Although we could parameterize binary search trees just with respect to a total order given by the theory `TOSET<`, we specify here the version of search trees containing in the nodes pairs formed by a key and its associated contents, so that we think of search trees as dictionaries. The search tree structure is with respect to a strict total order on keys, but contents can be over an arbitrary sort. When we insert a pair $\langle K, C\rangle$ and the key K already appears in the tree in a pair $\langle K, C'\rangle$, insertion takes place by combining the contents C' and C. This combination can be replacing the first with the second, just forgetting the second, addition if the trees are used to implement multisets and the Cs represent multiplicities, etc. Therefore, as part of the requirement parameter theory for the contents we will have an associative binary operation `combine` on the sort `Contents`. Notice how the two parameter theories on which the specification depends are separated by `|`.

In principle, a pair construction should be enough to put together the information that we have just described, but we will import the module for search trees in the specifications of more complex data structures that happen to be particular cases of search trees, such as AVL and red-black trees. In those cases, it is important to add new information in the nodes: for the AVL trees, one needs the depth of the tree hanging in each node, while for red-black trees one needs the appropriate node color. Therefore, taking this into account, it is important to define a datatype that is *extensible*, and we have considered *records* for this, defined in the module `RECORD`. A record is defined as a collection (with an associative and commutative union operator denoted by `_,_`) of pairs consisting of a field name and an associated value. After importing the binary trees instantiated with a `Record` view, we define the fields for keys (with syntax `key:_`) and for contents (with syntax `contents:_`), together with corresponding projection operations that extract the appropriate values from the record. Notice how these operations, as well as the operations on trees, can be applied to records that have more fields, unknown yet at this time, by using a variable `Rec` that takes care of "the rest of the record."

This construction of adding fields to records is modifying the data in the sort `Record`, which are the data in the nodes of the trees, and thus the trees themselves. For this reason, we have imported the module `BIN-TREE` (after instantiating it with the view `Record`) in `including` mode.

In addition to operations for insertion and deletion, we have a `lookup` operation that returns the contents associated to a given key, when the key appears in the tree. However, this last operation is partial,f because it is not defined when the key does not appear in the tree; this error state cannot be handled by means of a subsort, because the partiality condition depends on the concrete values of the arguments. We have used a *partial operator declaration* by stating that the result of the `lookup` operator is in the *kind* `[Y@Contents]` associated to the sort `Y@Contents` (notice the square brackets around the sort name to denote the corresponding kind). One can think of a kind as an error "supersort" where, in addition to correct well-formed terms, there are undefined or error terms. Instead of having at the level of kinds just a term that does not reduce, we have declared a special value `not-found` of kind `[Y@Contents]` which is returned by the `lookup` operation when the key does not appear in the tree.

These operations are specified as usual, by structural induction, and in the non-empty case by comparing the given key K with the key in the root of the tree and distinguishing the three cases according to whether K is smaller than, equal to, or bigger than the root key.

```
(fth CONTENTS is
   sort Contents .
   op combine : Contents Contents -> Contents [assoc] .
 endfth)

(fmod RECORD is
   sorts Record .
   op null : -> Record [ctor] .
   op _`,_ : Record Record -> Record [ctor assoc comm id: null] .
 endfm)

(view Record from TRIV to RECORD is
   sort Elt to Record .
 endv)

(fmod SEARCH-TREE(X :: TOSET< | Y :: CONTENTS) is
   including BIN-TREE(Record) .
   sorts SearchTree(X | Y) NeSearchTree(X | Y) .
   subsorts NeSearchTree(X | Y) < SearchTree(X | Y) < BinTree(Record) .
   subsort NeSearchTree(X | Y) < NeBinTree(Record) .

   --- Construction of the record used as node in binary search trees.
   op key:_ : X@Elt -> Record [ctor] .
   op contents:_ : Y@Contents -> Record [ctor] .
   op not-found : -> [Y@Contents] .
   op key : Record -> X@Elt .
   op contents : Record -> Y@Contents .
   var Rec : Record .   var K : X@Elt .   var C : Y@Contents .
   eq key(Rec,key: K) = K .
   eq contents(Rec,contents: C) = C .

   --- Operations for binary search trees.
   op insert : SearchTree(X | Y) X@Elt Y@Contents -> SearchTree(X | Y) .
   op lookup : SearchTree(X | Y) X@Elt -> [Y@Contents] .
   op delete : SearchTree(X | Y) X@Elt -> SearchTree(X | Y) .
   ops min max : NeSearchTree(X | Y) -> Record .
   var Rec' : Record .                 vars L R : SearchTree(X | Y) .
```

```
  vars L' R' : NeSearchTree(X | Y) .   var C' : Y@Contents .

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

  eq insert(empty, K, C) = empty [key: K,contents: C] empty .
  eq insert(L [Rec,key: K,contents: C] R, K, C') =
     L [Rec,key: K, contents: combine(C, C')] R .
  ceq insert(L [Rec] R, K, C) = insert(L, K, C) [Rec] R if K < key(Rec) .
  ceq insert(L [Rec] R, K, C) = L [Rec] insert(R, K, C) if key(Rec) < K .

  eq lookup(empty, K) = not-found .
  eq lookup(L [Rec,key: K,contents: C] R, K) = C .
  ceq lookup(L [Rec] R, K) = lookup(L, K) if K < key(Rec) .
  ceq lookup(L [Rec] R, K) = lookup(R, K) if key(Rec) < K .

  eq delete(empty, K) = empty .
  ceq delete(L [Rec] R, K) = delete(L, K) [Rec] R if K < key(Rec) .
  ceq delete(L [Rec] R, K) = L [Rec] delete(R, K) if key(Rec) < K .
  eq delete(empty [Rec,key: K,contents: C] R, K) = R .
  eq delete(L [Rec,key: K,contents: C] empty, K) = L .
  eq delete(L' [Rec,key: K,contents: C] R', K) =
      L' [min(R')] delete(R', key(min(R'))) .

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

3.10 AVL trees

It is well-known that in order to have better efficiency on search trees one has to keep them balanced. One nice solution to this problem is provided by AVL trees; these are binary search trees satisfying the additional constraint in each node that the difference between the depth of both children is at most one. This constraint guarantees that the depth of the tree is always logarithmic with respect to the number of nodes, thus obtaining a logarithmic cost for the operations of search, lookup, insertion and deletion, assuming that the last two are implemented in such a way that they keep the properties of the balanced tree. As we have already anticipated in Section 3.9, it is convenient to have in each node as additional data the depth of the tree having this node as root, so that comparing the depths of children to check the balance property of the AVL trees becomes very quick. This is accomplished by importing the module

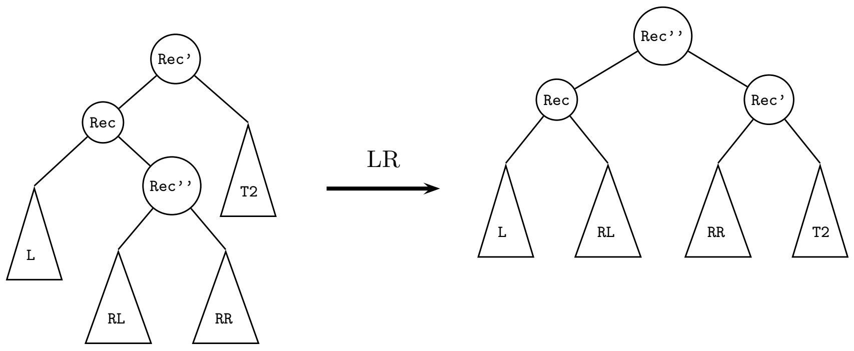

Fig. 1. Left-right rotation in an AVL tree.

SEARCH-TREE of search trees and adding a depth field to the record, together with the corresponding projection.

The sort AVL of AVL trees is a subsort of the sort SearchTree of search trees, defined by means of additional membership assertions (recall that we have already used this technique to define SearchTree as a subsort of BinTree in Section 3.9); in the specification below, just two memberships are enough, one for the empty tree and the other for non-empty AVL trees. Notice the use of the *symmetric difference* operator sd on natural numbers; the result of this operation applied to two natural numbers is the result of subtracting the smallest from the biggest of the two.

For lookup we use the same operation as for search trees, imported from the module SEARCH-TREE; on the other hand, insertion and deletion have to be redefined so that they keep the AVL properties. They work as in the general case, by comparing the given key with the one in the root, but the final result is built by means of an auxiliary join operation that checks that the difference between the depths of the two children is less than one, using again the symmetric difference operator sd; when this is not the case, appropriate *rotation* operations are invoked. It is enough to have a left rotation lRotate and a right rotation rRotate. This is quite similar to the typical imperative or object-oriented versions of these operations [9, 2, 16]. For example, the second equation for lRotate is illustrated in Figure 1. In the specification below we do not show the equations for deleteAVL and rRotate, which can be found in [10].

```
(fmod AVL(X :: TOSET< | Y :: CONTENTS) is
   including SEARCH-TREE(X | Y) .
   --- We add a new field to the node's record.
   op depth:_ : Nat -> Record [ctor] .
   op depth : Record -> Nat .
   var N : Nat .   vars Rec Rec' Rec'' : Record .
   eq depth(Rec,depth: N) = N .
```

```
  --- AVL trees memberships and operations.
  sorts NeAVL(X | Y) AVL(X | Y) .
  subsorts NeAVL(X | Y) < AVL(X | Y) < SearchTree(X | Y) .
  subsorts NeAVL(X | Y) < NeSearchTree(X | Y) .
  vars L R RL RR T1 T2 : AVL(X | Y) .

  mb empty : AVL(X | Y) .
  cmb L [Rec] R : NeAVL(X | Y) if L [Rec] R : SearchTree(X | Y) /\
                                  sd(depth(L),depth(R)) <= 1 /\
                            1 + max(depth(L),depth(R)) = depth(Rec) .

  op insertAVL : X@Elt Y@Contents AVL(X | Y) -> NeAVL(X | Y) .
  op deleteAVL : X@Elt AVL(X | Y) -> AVL(X | Y) .
  op depthAVL : AVL(X | Y) -> Nat .
  op buildAVL : AVL(X | Y) Record AVL(X | Y) -> AVL(X | Y) .
  op join : AVL(X | Y) Record AVL(X | Y) -> AVL(X | Y) .
  op lRotate : AVL(X | Y) Record AVL(X | Y) -> AVL(X | Y) .
  op rRotate : AVL(X | Y) Record AVL(X | Y) -> AVL(X | Y) .
  vars K K' : X@Elt .   vars C C' : Y@Contents .

  eq insertAVL(K, C, empty) =
       buildAVL(empty,(depth: 0,key: K, contents: C),empty) .
  eq insertAVL(K, C, L [Rec,key: K, contents: C'] R) =
       L [Rec,key: K, contents: combine(C,C')] R .
  ceq insertAVL(K, C, L [Rec] R) = join(insertAVL(K,C,L), Rec, R)
      if K < key(Rec) .
  ceq insertAVL(K, C, L [Rec] R) = join(L, Rec, insertAVL(K,C,R))
      if key(Rec) < K .

  eq depthAVL(empty) = 0 .
  eq depthAVL(L [Rec,depth: N] R) = N .

  eq buildAVL(T1, (Rec',depth: N), T2) =
       T1 [Rec',depth: (max(depthAVL(T1),depthAVL(T2)) + 1)] T2 .

  ceq join(T1, Rec, T2) = buildAVL(T1, Rec, T2)
        if sd(depthAVL(T1),depthAVL(T2)) <= 1 .
  ceq join(T1, Rec, T2) = lRotate(T1, Rec, T2)
        if depthAVL(T1) = depthAVL(T2) + 2 .
  ceq join(T1, Rec, T2) = rRotate(T1, Rec, T2)
        if depthAVL(T1) + 2 = depthAVL(T2) .

  ceq lRotate(L [Rec] R,Rec',T2) = buildAVL(L,Rec,buildAVL(R,Rec',T2))
      if depthAVL(L) >= depthAVL(R) .
  ceq lRotate(L [Rec] (RL [Rec''] RR), Rec', T2) =
        buildAVL(buildAVL(L, Rec, RL), Rec'', buildAVL(RR, Rec', T2))
      if depthAVL(L) < depthAVL(RL [Rec''] RR) .
endfm)
```

3.11 2-3-4 trees

Other solutions to the problem of keeping balanced search trees are provided by *2-3 trees*, which are not treated here, and *2-3-4 trees*, whose specification we consider in this section. This kind of search trees generalizes binary search trees to a version of general trees of degree 4, so that a non-leaf node can have 2, 3 or 4 children. The number of values in the node depends on the number of children; for example, there are two different values (let us call N1 the smallest of the two, and N2 the greatest) in the node when it has three children. Moreover, the values in the children are well organized with respect to the values in the node; in the same example, all the values in the first child must be smaller than N1, all the values in the second child must be bigger than N1 and smaller than N2, and all the values in the third child must be bigger than N2. Furthermore, the children must have exactly the same depth, and recursively they have to satisfy the same properties. As expected, all of these properties can be stated by means of memberships assertions.

Since these trees need a different set of constructors, they have no direct relationship to binary search trees. Also, in order to simplify the presentation we just parameterize the specification with respect to the theory TOSET<, that is, we consider only values in the nodes, instead of keys and associated values as we did in previous sections.

The specification of the search operation is immediate. Although the main ideas of insertion are quite simple, the details of the implementation become much lengthier than expected, requiring several auxiliary operations and several equations to treat the different cases arising from combining the different constructors. Even worse is the implementation of deletion, which needs a *zillion* of equations to deal with all possible cases. No wonder most textbooks avoid presenting these details and leave them as a challenging programming project!

Here we only show the memberships that specify 2-3-4 trees; the remaining details are available in [10].

```
sort Ne234Tree?(T) 234Tree?(T) Ne234Tree(T) 234Tree(T) .
subsort Ne234Tree?(T) < 234Tree?(T) .
subsort Ne234Tree(T) < 234Tree(T) < 234Tree?(T) .
subsort Ne234Tree(T) < Ne234Tree?(T) .

op empty234 : -> 234Tree?(T) [ctor] .
op _`[_`]_ : 234Tree?(T) T@Elt 234Tree?(T) -> Ne234Tree?(T) [ctor] .
op _<_>_<_>_ : 234Tree?(T) T@Elt  ...  -> Ne234Tree?(T) [ctor] .
op _`{_`}_`{_`}_`{_`}_ : 234Tree?(T) T@Elt ... -> Ne234Tree?(T) [ctor] .
vars N N1 N2 N3 : T@Elt .
var 234TL 234TLM 234TC 234TRM 234TR : 234Tree(T) .

mb empty234 : 234Tree(T) .
cmb 234TL [ N ] 234TR : Ne234Tree(T) if greaterKey(N,234TL) /\
    smallerKey(N,234TR) /\ depth(234TL) = depth(234TR) .
cmb 234TL < N1 > 234TC < N2 > 234TR : Ne234Tree(T)
    if N1 < N2 /\ greaterKey(N1,234TL) /\ smallerKey(N1,234TC) /\
                  greaterKey(N2,234TC) /\ smallerKey(N2,234TR) /\
       depth(234TL) = depth(234TC) /\ depth(234TC) = depth(234TR) .
cmb 234TL { N1 } 234TLM { N2 } 234TRM { N3 } 234TR : Ne234Tree(T)
    if N1 < N2 /\ N2 < N3 /\ greaterKey(N1,234TL) /\
       smallerKey(N1,234TLM) /\ greaterKey(N2,234TLM) /\
       smallerKey(N2,234TRM) /\ greaterKey(N3,234TRM) /\
       smallerKey(N3,234TR) /\ depth(234TL) = depth(234TLM) /\
       depth(234TL) = depth(234TRM) /\ depth(234TL) = depth(234TR) .
```

3.12 Red-black trees

Yet another solution to the problem of keeping search trees balanced are *red-black* search trees. These are standard binary search trees that satisfy several additional constraints that are related to a color (hence the name!) that can be associated to each node (in some presentations, to the edges). One can think of red-black trees as a binary representation of 2-3-4 search trees, and this provides helpful intuition.

Since the color is additional information in each node, we make again use of the record construction described in Section 3.9. Once more, memberships allow a faithful specification of all the constraints. All details can be found in [10].

3.13 Priority queues

The abstract specification of priority queues is very simple. It is parameterized with respect to the theory `TOSET<=`, because we allow repetitions and the priority is identified with each value. We have as constructors the constant `empty` and `insert`, that adds a new element to the priority queue. However, these constructors are not free because the order of insertion does not matter, the priority being the information that determines the actual order in the queue. This is made explicit in a "commutativity" equation for the `insert` operator, but this is not a standard commutativity equation for a binary operator with both arguments of the same sort, and thus it cannot be expressed as a `comm` attribute; in any case, it is not terminating, and therefore it has been stated as *non-executable* by means of the `nonexec` attribute (which is associated to the equation, and not to the operator).

We consider the version of priority queues in which the first element is the minimum. Both `findMin` and `deleteMin` are easily specified as total operations on non-empty priority queues by structural induction and in the second case by comparing the priorities of two elements.

Even though this is a very abstract specification, it is directly executable after instantiating appropriately the parameter.

```
(fmod PRIORITY-QUEUE(X :: TOSET<=) is
  sort NePQueue(X) PQueue(X) .
  subsort NePQueue(X) < PQueue(X) .
  op empty : -> PQueue(X) [ctor] .
  op insert : PQueue(X) X@Elt -> NePQueue(X) [ctor] .
  op deleteMin : NePQueue(X) -> PQueue(X) .
  op findMin : NePQueue(X) -> X@Elt .
  op isEmpty : PQueue(X) -> Bool .
  var PQ : PQueue(X) .  vars E F : X@Elt .
  eq insert(insert(PQ,E),F) = insert(insert(PQ,F),E) [nonexec] .
  eq  deleteMin(insert(empty,E)) = empty .
  ceq deleteMin(insert(insert(PQ,E),F)) =
       insert(deleteMin(insert(PQ,E)),F) if findMin(insert(PQ,E)) <= F .
  ceq deleteMin(insert(insert(PQ,E),F)) =
       insert(PQ,E) if findMin(insert(PQ,E)) > F .
  eq  findMin(insert(empty,E)) = E .
  ceq findMin(insert(insert(PQ,E),F)) =
       findMin(insert(PQ,E)) if findMin(insert(PQ,E)) <= F .
  ceq findMin(insert(insert(PQ,E),F)) =
       F if findMin(insert(PQ,E)) > F .
  eq isEmpty(empty) = true .
  eq isEmpty(insert(PQ,E)) = false .
endfm)
```

3.14 Leftist trees

Among the many different data structures implementing priority queues the most efficient are *heaps*, which can be defined (in the case of *min heaps*) as binary trees satisfying the additional constraints that the value in each node is smaller than (or equal to) the values in its children and moreover the tree is complete. If we forget the latter requirement and instead assign to each node a *rank* (also known as *minimum depth*) defined as the length of the rightmost path to a leaf, and require that the root of each left child has a rank bigger than or equal to the rank of the corresponding right child (that can be empty), we get the trees known as *leftist trees*. These trees implement priority queues with the same efficiency as standard heaps, and have the additional property that two leftist trees can be merged to obtain a leftist tree containing all elements in the two given trees in logarithmic time with respect to the total number of nodes.

As usual, the sort of leftist trees can be defined as a subsort of binary trees by means of appropriate membership assertions. In order to compare quickly the ranks of two nodes, we need to save in each node its rank (in the same way that we saved the depth in each node in AVL trees). Since we are importing binary trees, which are parameterized with respect to the theory `TRIV`, we first define in the module `TREE-NODE` the construction of pairs formed by an element and a natural number, and then instantiate the module `BIN-TREE` of binary trees with the *parameterized view* `Node(T)`. The view is parameterized because it still keeps as a parameter, as expected, the sort of the elements with a total order on top of which we are building the leftist trees.

The most important operation on this data structure is `merge` because both `insert` and `deleteMin` are easily defined from it. The operation `merge` is specified by structural induction on its arguments, with the help in the recursive case (when both arguments are non-empty) of an auxiliary operation `make` that takes care of putting the tree with bigger rank at its root to the left of the tree being built.

```
(fmod TREE-NODE(T :: TOSET<=) is
   protecting NAT .
   sort Node(T) .
   op n : Nat T@Elt -> Node(T) [ctor] .
 endfm)

(view Node(T :: TOSET<=) from TRIV to TREE-NODE(T) is
   sort Elt to Node(T) .
 endv)

(fmod LEFTIST-TREES(T :: TOSET<=) is
   protecting BIN-TREE(Node(T)) .
   sorts NeLTree(T) LTree(T) .
   subsorts NeLTree(T) < LTree(T) < BinTree(Node(T)) .
   subsorts NeLTree(T) < NeBinTree(Node(T)) .

   op rank : BinTree(Node(T)) -> Nat .
   op rankL : LTree(T) -> Nat .
   op insert : T@Elt LTree(T) -> NeLTree(T) .
   op deleteMin : NeLTree(T) -> LTree(T) .
   op findMin : NeLTree(T) -> T@Elt .
   op make : T@Elt LTree(T) LTree(T) -> LTree(T) .
   op merge : LTree(T) LTree(T) -> LTree(T) .
   vars NeTL NeTR : NeLTree(T) .              vars M N N1 N2 : Nat .
   vars T TL TR TL1 TR1 TL2 TR2 : LTree(T) .   vars X X1 X2 : T@Elt .

   mb empty : LTree(T) .
   mb empty [n(1, X)] empty : NeLTree(T) .
   cmb NeTL [n(1, X)] empty : NeLTree(T) if X < findMin(NeTL) .
   cmb NeTL [n(N, X)] NeTR : NeLTree(T) if rank(NeTL) >= rank(NeTR) /\
         X < findMin(NeTL) /\ X < findMin(NeTR) /\ N = 1 + rank(NeTR) .
   eq rank(empty) = 0 .
   eq rank(TL [n(N, X)] TR) = 1 + rank(TR) .
   eq rankL(empty) = 0 .
   eq rankL(TL [n(N,X)] TR) = N .
   eq merge(empty, T) = T .
   eq merge(T, empty) = T .
   eq merge(TL1 [n(N1, X1)] TR1, TL2 [n(N2, X2)] TR2) =
        if X1 < X2 then make(X1,TL1,merge(TR1, TL2 [n(N2,X2)] TR2))
                   else make(X2,TL2,merge(TL1 [n(N1,X1)] TR1,TR2)) fi .
   eq make(X, TL, TR) = if rankL(TL) >= rankL(TR)
                        then TL [n(rankL(TR) + 1,X)] TR
                        else TR [n(rankL(TL) + 1,X)] TL fi .
   eq insert(X,T) = merge(empty [n(1,X)] empty, T) .
   eq findMin(TL [n(N,X)] TR) = X .
   eq deleteMin(TL [n(N,X)] TR) = merge(TL,TR) .
 endfm)
```

4 Proving Properties

Mechanical reasoning about specifications in Maude is supported by the experimental ITP tool [3], a rewriting-based theorem prover that can be used to prove inductive properties of equational specifications. It is written in Maude, and it is itself an executable specification. A key feature of the ITP is its reflective design, that allows the definition of customized *rewriting strategies* different from Maude's default one; currently, this capability is being used to extend the ITP with decision procedures for arithmetic, lists, and combinations of theories.

The ITP is still very much work in progress: it can work with any module present in Maude's database but not with those introduced in Full Maude; in particular, at present it offers no support for parameterized specifications. Therefore, in the examples that follow we illustrate its use with the concrete modules `LIST` and `ORD-LIST` of *lists of natural numbers*, obtained from the specifications in Sections 3.3 and 3.4 by removing the parameter `X` and renaming the sort `X@Elt` as `Nat`.

4.1 Concatenation is associative

The most basic property the `LIST` specification should satisfy is that concatenation of lists is associative. After loading the ITP with the instruction `in xitp-tool` and initializing its database with `loop init-itp .`, the property can be proved with the following commands enclosed in parentheses:

```
(goal list-assoc : LIST |- A{L1:List ; L2:List ; L3:List}
 (((L1:List ++ L2:List) ++ L3:List) = (L1:List ++ (L2:List ++ L3:List))) .)
  (ind on L1:List .)
    (auto* .) --- base case []
    (auto* .) --- inductive step E : L
```

The first two lines introduce the desired goal: `list-assoc` is its name, `LIST` the module in which the goal is to be proved, and the symbol `A` (representing ∀) precedes a list of universally quantified variables. Notice that variables are always annotated with their types. Then we try to prove the property by structural induction on the first variable and the ITP generates a corresponding subgoal for each operator of range `List` that has been declared with the attribute `ctor`; in this case one for the empty list `[]` and another one for the constructor `_:_` (with its corresponding inductive hypothesis). Finally, both cases can be automatically proved with the command `auto*` that first transforms all variables into fresh constants and then rewrites the terms in both sides of the equation as much as possible.

4.2 Reverse is self-inverse

As another example we show that the result of applying the operation `rev` twice in a row to a list leaves it unchanged. Again, we try to prove the property by induction but this time the second `auto*` does not succeed and leaves us with the following subgoal to prove:

```
|- rev(rev(V0#1*List)++ V0#0*Elem :[])= V0#0*Elem : V0#1*List
```

This suggests proving the auxiliary lemma `conc-rev` below, which in turn requires a lemma to show that `[]` is the identity for concatenation and another one for associativity as above. Lemmas are introduced like goals but using the keyword `lem` and without the module's name; all three are straightforwardly proved by structural induction, and then the pending subgoal can be discharged with a final `auto*`.

```
(goal list-rev : LIST |- A{L:List}((rev(rev(L:List))) = (L:List)) .)
  (ind on L:List .)
    (auto* .) --- []
    (auto* .) --- E : L
(lem conc-id : A{L:List} ((L:List ++ []) = (L:List)) .)
  (ind on L:List .)
    (auto* .)
    (auto* .)
(lem conc-assoc : A{L1:List ; L2:List ; L3:List}
 (((L1:List ++ L2:List) ++ L3:List) = (L1:List ++ (L2:List ++ L3:List))) .)
  (ind on L1:List .)
    (auto* .)
    (auto* .)
(lem conc-rev : A{L1:List ; L2:List}
     ((rev(L1:List ++ L2:List)) = (rev(L2:List) ++ rev(L1:List))) .)
  (ind on L1:List .)
    (auto* .)
    (auto* .)
(auto* .)
```

4.3 Ordered lists

The final example we consider is showing that the typing assigned to the operation `insert-list` in `ORD-LIST` is indeed correct; that is, that inserting a new element to an ordered list returns an ordered list. Note that in doing so we use an equationally defined predicate `sorted` instead of a membership assertion `_:OrdList`; again, this is due to the ITP's current restrictions.

The proof proceeds as usual by structural induction and, as in the previous example, we cannot discharge the second case, corresponding to a list of the form `E : L`, with a simple `auto*`. Note that this time, before proving the auxiliary lemma it is also necessary to distinguish the case in which the value `N` being inserted is less than or equal to `E`, from that in which it is not. This is done in the ITP with the `split` command: `N*` and `V0#0*Nat` are the constants to which `auto*` has transformed the original variables `N` and `E`. The rest of the proof follows the same pattern as the previous ones.

```
(goal SORTED : ORD-LIST |- A{L:List ; N:Nat}
   (((sorted(L:List)) = (true)) =>
    ((sorted(insert-list(L:List, N:Nat))) = (true))) .)
```

```
  (ind on L:List .)
    (auto* .)
    (auto* .)
    (split on (N*Nat <= V0#0*Nat) .)
    (auto* .)
    (auto* .)
(lem SORTED2 : A{N:Nat ; M:Nat ; L:List}
 (((sorted(N:Nat : L:List)) = (true) & (N:Nat <= M:Nat) = (true)) =>
  ((sorted(N:Nat : insert-list(L:List, M:Nat))) = (true))) .)
  (ind on L:List .)
    (auto* .)
    (auto* .)
    (split on (M*Nat <= V1#0*Nat) .)
    (auto* .)
    (auto* .)
(auto* .)
```

5 Ongoing and future work

As mentioned in the introduction, we consider this work as a first step to develop a set of basic data structures that could eventually lead to a shared library; now, other users' feedback will be necessary to improve this contribution. For such a library, one would expect an implementation as efficient as possible; however, there is a clear tension between this goal and that of using the specifications to illustrate programming in Maude.

There are a number of techniques to improve the efficiency in Maude, like the systematic use of *unconditional* equations with the help of the `if_then_else_fi` operator and the `owise` attribute. We have rewritten several of our examples using these techniques and obtaining a considerable gain in efficiency, as expected. They have the drawback of making formal reasoning about the specifications much more difficult or even impossible with the current version of the ITP. There is also the tradeoff between efficiency and very precise typing by means of memberships; these require typechecking during execution. If one is willing to work with less refined types, as is the case with other functional languages, one can forget about most of the memberships that appear in our specifications, thus obtaining another considerable speedup. Finally, one must distinguish also between the efficiency in the Full Maude prototype that works at the metalevel and the one in the Core Maude system implemented in C++; the transfer of parameterization techniques to the core level will help in this respect. In particular, according to the limited set of tests that we have run, Core Maude's performance in executing non-parametric specifications is close to that of GHCi for Haskell 98 [8], whereas Full Maude can be as much as three times slower.

Concerning the proof of properties, in collaboration with Manuel Clavel we have managed to prove more complex ones, including the correctness of the sorting operations `mergesort` and `quicksort`. However, we are still finding our way with proofs of basic properties about search trees. In the future, we would like to consider more complex relationships such as those between 2-3-4 and red-black trees.

Acknowledgments. We warmly thank Ricardo Peña for our discussions on functional implementations of data structures, Manuel Clavel for his work on the ITP and his help in understanding its idiosyncrasies, José Meseguer for his encouragement, and the referees for their detailed and helpful comments.

References

1. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
2. F. M. Carrano and J. J. Prichard. *Data Abstraction and Problem Solving with C++. Third Edition.* Addison-Wesley, 2002.
3. M. Clavel. The ITP Tool. `http://geminis.sip.ucm.es/~clavel/itp`, 2004.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *A Maude Tutorial.* Computer Science Laboratory, SRI International, 2000. `http://maude.cs.uiuc.edu/papers`.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 2003, Proceedings*, LNCS 2706, pages 76–87. Springer, 2003.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1)*, March 2004. `http://maude.cs.uiuc.edu/manual`.
7. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language.* PhD thesis, Universidad de Málaga, 1999.
8. The GHC team. *The Glasgow Haskell Compiler*, March 2004. `http://www.haskell.org/ghc`.
9. E. Horowitz and S. Sahni. *Fundamentals of Data Structures in Pascal. Fourth Edition.* Computer Science Press, 1994.
10. N. Martí-Oliet, M. Palomino, and A. Verdejo. Data structures in Maude. `http://maude.sip.ucm.es/datastructures`, 2004.
11. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, LNCS 1376, pages 18–61. Springer, 1998.
12. C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.
13. M. Palomino, N. Martí-Oliet, and A. Verdejo. Playing with Maude. In *RULE'04 Fifth International Workshop on Rule-Based Programming. Proceedings*, Electronic Notes in Theoretical Computer Science, pages 4–19. Elsevier, 2004.
14. R. Peña. *Diseño de Programas. Formalismo y Abstracción. Segunda Edición.* Prentice Hall, 1998.
15. F. Rabhi and G. Lapalme. *Algorithms. A Functional Programming Approach.* Addison-Wesley, 1999.
16. M. A. Weiss. *Data Structures and Problem Solving Using Java.* Addison-Wesley, 1998.